DATACOMPUTER PROJECT TECHNICAL REPORT

Computer Corporation of America

Prepared for:

Army Research Office-Durham
Advanced Research Projects Agency

28 February 1974

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

DATACOMPUTER PROJECT
TECHNICAL REPORT

August 1, 1973 to February 28, 1974

i.

Abstract

The datacomputer system is being designed as a large-scale
data storage utility to be accessed from remote computers on
the Arpanet and, potentially, on other networks. The develop-
ment is phased, with each successive release of the system
offering increased capabilities to users. During the present
reporting period, the second release of the system became
operational. This release, while still primitive in many
respects, is beginning to provide experience with actual
applications and user programs. The present document is the
final technical report under Contract No. DAHC04-71-C-0011.
The project is continuing under Contract No. MDA903-74-C-0225.

Table of Contents

Figures

1. Overview

1.1 Review of Basic Concepts

The goal of the project continues to be the development of
a shared, large-scale data storage utility, to serve the
needs of the Arpanet community.

The system under development will make it possible to store
within the network such files as the ETAC Weather File or
the NMRO Seismic Data File, which are measured in hundreds
of billions of bits, and to make arbitrarily selected parts
of these files available within seconds to sites requesting
the information. The system is also intended to be used as
a centralized facility for archiving data, for sharing data
among the various network hosts, and for providing inexpensive
on-line storage to sites which need to supplement their local
capability.

Logically, the system can be viewed as a closed box which
is shared by multiple external processors, and which is
accessed in a standard notation, "datalanguage" (see Fig. 1).
The processors can request the system to store information,
change information already stored in the system, and retrieve
stored information. To cause the datacomputer to take action,
the external processor sends a "request" expressed in data-
language to the datacomputer, which then performs the desired
data operations.

From the user's point of view the datacomputer is a remotely-
located utility, accessed by telecommunications. It would be
impractical to use such a utility if, whenever the user wanted
to access or change any portion of his file, the entire file
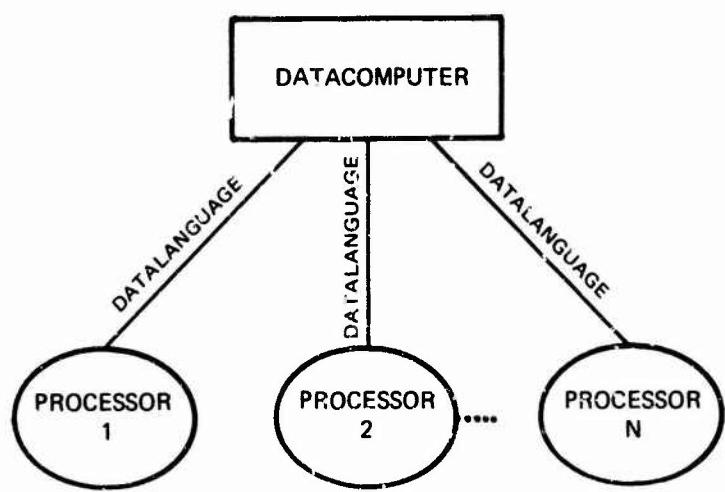
Figure 1. Logical View of Datacomputer

I. --2--

had to be transmitted to him. Accordingly, data management
functions (information retrieval, file maintenance, backup,
access security, creation of direct and inverse files, main-
tenance of file directories, etc.) are performed by the data-
computer system itself. The user sends a "request", which
causes the proper functions to be executed at the datacomputer
without requiring entire files to be shipped back and forth.

The hardware of the system is shown in overview in Fig. 2
and in greater detail in Fig. 3.

The program for the system processor handles the interactions
with the network hosts and is designed to control up to three
levels of storage: primary (core), secondary (disk), and
tertiary mass storage. Currently, the CCA facility is operating
with primary and secondary storage only, with the addition of
tertiary storage planned for 1975. Installation of a tertiary
storage module will leave datalanguage unchanged, and will
therefore be imperceptible to users of the system (except
insofar as it affects performance and the total storage capacity
available for data).

In addition to using the dedicated equipment at CCA it is
planned that datacomputer service will also make use of
hardware resources located at NASA/Ames, using CCA software.
The two sites will provide mutual backup for one another,
thereby guarding against accidental loss of data and providing
for satisfactory uptime of the overall service.
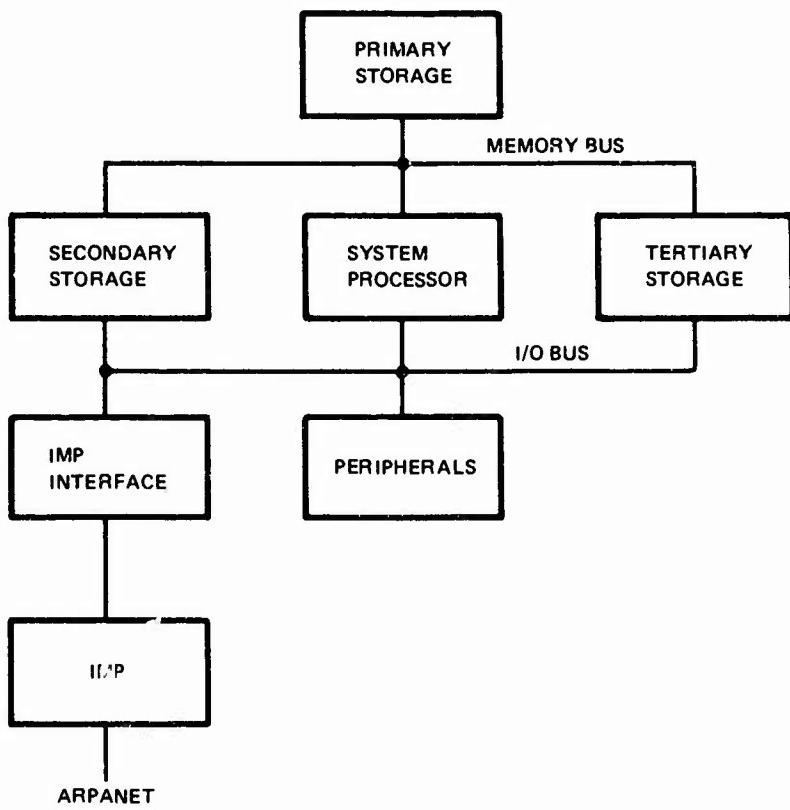
Figure 2. Hardware Overview of System

I-4-
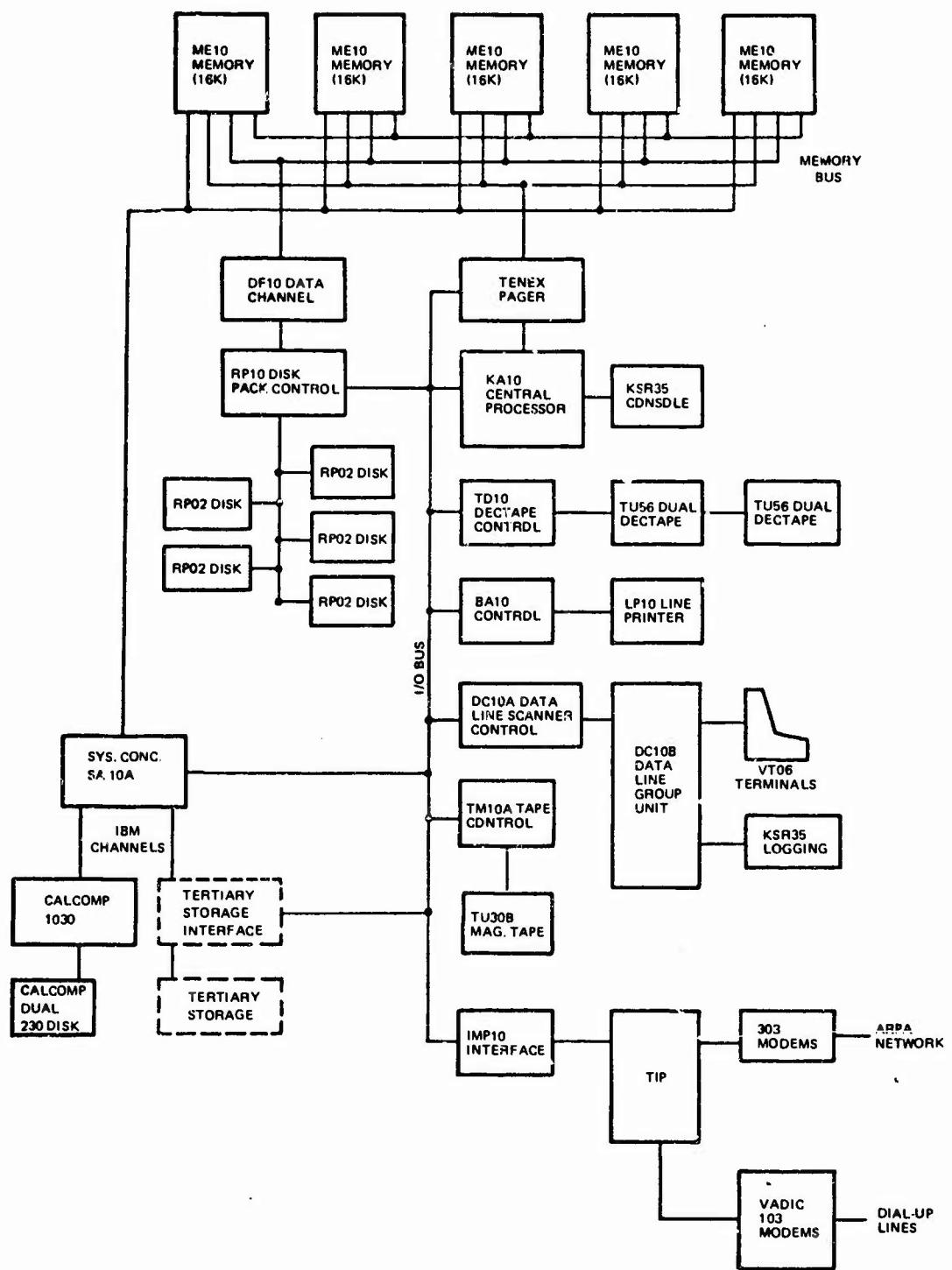
Figure 3. Hardware Block Diagram - CCA Installation
(Equipment in dashed outline is planned for 1975)

1.2 Status of Project

The present reporting period brings the datacomputer project to the end of its first phase, under Contract No. DAHC04-71-C-0011, during which an initial operating capability was achieved. The next phase of the work is being performed under Contract No. MDA903-74-C-0225.

During this reporting period, Version 0/9.7 of the datacomputer system was completed. This is the second version of the system to offer datacomputer services on the Arpanet. In this new version the most serious restriction of the previous version has been lifted, namely the restriction that elementary data elements must consist of fixed-length ASCII strings. Version 0/9.7 allows variable-length data and binary data. In addition, it includes an optimized file load technique. New features are described in more detail in Section 3.

Version 0/9.7 is an "intermediate" release. Work has continued on Version 0/10, the next major release.

Currently, only disk storage is available to the system. A Calcomp Dual 230 disk was installed during the fourth quarter of 1973. This increases the total disk storage of the CCA installation from 9×10^8 bits to 2.34×10^9 bits. The additional disk was necessary in order to keep all of the data currently stored on the datacomputer on-line.

Plans call for the addition of tertiary storage in 1975. Since datalanguage is device-independent, these increases in storage capacity will not affect the user programs running on network hosts.

Under the new contract, the datacomputer project now enters
Phase II of its development. Phase II has two goals: (1) the
operation of a large-scale data utility service for the
Arpanet, based on previous developments in the project and
(2) a continuing program of research and development in network
data handling, with emphasis on enhancements to the service.

2. Design Activities

2.1 Datalanguage

Work on a major design iteration of datalanguage continued.
This design effort incorporates new ideas that have developed
as a result of experience gained in implementation and service,
through further study of user requirements and work with
potential users, and through investigation of other work in
the data management field.

The design activity resulted in Datacomputer Project Working
Paper No. 8, "Further Datalanguage Design Concepts", dated
December 15, 1973 (see Appendix). This paper presents the
concepts and preliminary results of the current datalanguage
design effort. It discusses global considerations for an
expanded datalanguage design, specific issues which have been
addressed by the design team, and presents a working model for
the new language which was developed using the EL1 language.
The working paper discusses such data description issues as
the specification of basic and aggregate data objects; general-
ized relationships between data objects, virtual data objects,
and the representation of ordered sets. Issues of internal
representation, language mechanisms for data reference, control
of program flow, and the basic operations used for manipulating
data objects are outlined. The problems of data integrity and
privacy are also addressed.

The emphasis in the working paper is on solutions and options
of solutions. It does not contain a completed design. Work
continues toward a detailed specification for the new data-
language.

2.2 Software System

The current design of the software system basically corresponds to the design documented in Working Paper No. 5 (February 29, 1972).

3. Software Implementation

During this period, Version 0/9.7 of the datacomputer system
was completed and began offering service on the Arpanet.
Work on Version 0/10, the next major release, was in progress.
The main features of each of these systems are described in
this section.

3.1 Request Handler

Version 0/9.7 handles a larger range of data and file formats
than the previous version. In addition to 7-bit ASCII, the
simple data types now include 8-bit ASCII and uninterpreted
bytes or byte strings. The user specifies a byte size less
than or equal to 36. A string may be variable-length with
either a preceding one-byte count or, if it is in a PORT, a
trailing "punctuation" character (i.e., end-of-record, end-of-
block or end-of-file marker). In Version 0/10, a variable-
length string can be terminated by a user-specified delimiter.

A file or port may now be any aggregate or simple data type;
formerly a file or port was required to be a list.

A login command is provided in Version 0/9.7. It provides a
context for recognition of container names. In Version 0/10,
a file-level security system will be integrated with the login
command.

Version 0/9.7 includes several optimizations that were not
in previous versions. One of these is the deferred mode for
loading or updating files which have inversions. First, the
update of the base is completed. (The base is the part of the
file which contains only the user's data.) Then, inversion
update entries are sorted, thus minimizing disk accesses to
inversion pages.

Two optimizations that result in faster execution time were
added to the compiler. It should be noted that this kind of
optimization reflects the bias in datacomputer development
towards handling large files, where compilation time is rela-
tively insignificant and efficient execution is essential. One
of these optimizations results in run-time handling of a number
of containers as a single container when there is no need to
parse them individually. The second optimization is the addition
of the IG (Instruction Generator) module, which scans the tuple
table and assembles PDP-10 code which will simulate tuples. The
tuple code has been written to facilitate simple optimization
by IG. This includes suppressing some redundant loads of
temporary registers, "de-indexing" indexed XCT's (and de-XCTing
them), eliminating extraneous transfers (eligibility indicated
in the tuple code by the use of the 'JUMPA' instruction, instead
of 'JRST'), and suppressing all run-time conditional jumps and
skips based on information available at IG-time.

In Version 0/10 members of inner level lists will be valid
in boolean expressions. It will also be possible to invert
members of inner level lists. However, only EQ can be evaluated
using the inversion; evaluation of NE will still require a
sequential search of the data.

3.2 Services
The datacomputer can now use a 3330-type disk as its storage
medium. It is treated as special disk, that is, disk that is
completely under datacomputer control rather than Tenex control.
A JSYS has been added to Tenex for I/O to the 3330. However,
the disk is not part of the normal Tenex page space.

The directory data structures have been changed. Indirect
pointers were deleted, and the name blocks were simplified.
This is largely in preparation for adding security features
in Version 0/10.

4. Network Service

4.1 User Programs

At the end of the previous reporting period, the datacomputer
system achieved its initial operating capability on the Arpanet.
Two user programs for accessing the datacomputer in datalanguage
from Arpanet hosts were provided by CCA.

One of the user programs is SMART, which allows a user at a
terminal to access the datacomputer. SMART has information
about datacomputer files assembled into it. It generates data-
language for outputting subsets of these files. It also handles
all datacomputer protocols, Network communication and terminal
I/O. It was used during this reporting period to access two
ETAC files, one a file of weather observations, and the other
a station library.

FORPAC is a set of subroutines that give FORTRAN programs
access to the datacomputer. Like SMART, it has information
about datacomputer files, and it uses this information to
generate datalanguage for retrieving subsets of the files. The
calling FORTRAN program can then operate on the data. FORPAC
was used on an experimental basis by ETAC for a precipitation
study.

Appendix

Working Paper No. 8, "Further Datalanguage
Design Concepts", December 15, 1973

Further Datalanguage Design Concepts

I-15

Further Datalanguage Design Concepts

Richard Winter
Jeffrey Hill
Warren Greiff

Acknowledgement

I.17

CONTENTS

1. Introduction

1.1 The Datacomputer System

The datacomputer is a large-scale data utility system, offering data storage and data management services to other computers.

The datacomputer differs from traditional data management systems in several ways.

First, it is implemented on dedicated hardware, and comprises a separate computing system specialized for data management.

Second, the system is implemented on a large scale. Data is intended to be stored on mass storage devices, with capacities in the range of a trillion bits. Files on the order of one hundred billion bits are to be kept online.

Third, it is intended to support sharing of data among processes operating in diverse environments.   That is, the programs which share a given data base may be written in different languages, execute on different hardware under different operating systems, and support end users with radically different requirements. To enable such shared use of a data base, transformations between various hardware representations and data structuring concepts must be achieved.

Finally, the datacomputer is designed to function smoothly as a component of a much larger system: a computer network.   In a computer network, the datacomputer is a node specialized for data management, and acting as a data utility for the other nodes.   The Arpanet, for which the datacomputer is being developed, is an international network which has over 60 nodes.   Of these, some are presently specialized for terminal handling, others are specialized for computation (e.g., the ILLIAC IV), some are general purpose service nodes (e.g., MULTICS) and one (CCA) is specialized for data management.

1.2 Datalanguage

Datalanguage is the language in which all requests to the datacomputer are stated.   It includes facilities for data description and creation, for retrieval of or changes to stored data, and for access to a variety of auxiliary facilities and services. In datalanguage it is possible to specify any operation the datacomputer is capable of performing. Datalanguage is the only language accepted by the datacomputer and is the exclusive means of access to data and services.

1.3 Present Design Effort

We are now engaged in developing complete specifications for datalanguage; this is the second iteration in the language design process.

A smaller, initial design effort developed some concepts and priniciples which are described in the third working paper in this series. These have been used as the basis of software implementation, resulting in an initial network service capability. A user manual for this system was published as working paper number 7.

As a result of experience gained in implementation and service, through further study of user requirements and work with potential users, and through investigation of other work in the data management field, quite a few ideas have been developed for the improvement of datalanguage. These are being assimilated into the language design in the iteration now in progress.

When the language design is complete, it will be incorporated into the existing software (requiring changes to the language compiler, but having little impact on the rest of the system).

Datacomputer users will first have access to the new language during 1975.

1.4 Purpose of this Paper

This paper presents concepts and preliminary results, rather than a completed design. There are two reasons for publishing now.

The first is to provide information to those planning to use the datacomputer. They may benefit from knowledge of our intentions for development.

The second is to enable system and language designers to comment on our work before the design is frozen.

1.5 Organization of the Paper

The remainder of the paper is divided into four sections.

Section 2 discusses the most global considerations for language design. This comprises our view of the problem; it has influenced our work to date and will determine most of our actions in completion of the design. This section provides background for section 3, and reviews some material that will be familiar to those who have been following our work closely.

Section 3 discusses some of the specific issues we have worked on. The emphasis is on solutions and options for solution.

In sections 2 and 3 we are presenting our "top-down" work: this is the
thinking we   have done based cn known requirements and our conception of the
desirable properties of datalanguage.

We have also been working from the opposite end, developing the primitives
from which to construct the language. Section 4 presents our work in this
area: a model datacomputer which will ultimately provide a precise semantic
definition of datalanguage. Section 4 explains that part of the model which
is complete, and relates this to our other work.

Section 5 discusses work that remains, both on the model and in our top-down
analysis.

2. Considerations for Language Design

2.1 Introduction

Data management is the task of managing data as a resource, independent of hardware and applications programs. It can be divided it into five major sub-tasks:

      (1) creating databases in storage,
      (2) making the data available (e.g., satisfying queries),
      (3) maintaining the data as information is added, deleted and modified,
      (4) assuring the integrity of the data (e.g., through backup and recovery systems, through internal consistency checks),
      (5) regulating access, to protect the databases, the system, and the privacy of users.

These are the major data-related functions of the datacomputer; while the system will ultimately provide other services (such as accounting for use, monitoring performance) these are really auxiliary and common to all service facilities.

This section presents global considerations for the design of datalanguage, based on our observations about the problem and the environment in which it is to be solved. The central problem is data management, and the datacomputer shares the same goals as many currently available data management systems. Several aspects of the datacomputer create a unique set of problems to be solved.

2.2 Hardware Considerations

2.2.1 Separate Box

The datacomputer is a complete data management utility in a separate, closed box. That is, the hardware, the data and the data management software are segregated from any general-purpose processing facilities. There is a separate installation dedicated to data management. Datalanguage is the only means users have for communicating with the datacomputer and the sole activity of the datacomputer is to process datalanguage requests.

Dedicating hardware provides an obvious advantage: one can specialize it for data management. The processor(s) can be modified to have data management "instructions"; common low-level software functions can be built into the hardware.

A less obvious, but possibly more significant, advantage is gained from the separateness itself. The system can be more easily protected. A

fully-developed datacomputer on which there is only maintenance activity can provide a very carefully controlled environment. First, it can be made as physically secure as required. Second, it needs to execute only system software developed at CCA; all user programs are in a high-level language (datalanguage) which is effectively interpreted by the system. Hence, only datacomputer system software processes the data, and the system is not very vulnerable to capture by a hostile program. Thus, since there is the potential to develop data privacy and integrity services that are not available on general-purpose systems, One can expect less difficulty in developing privacy controls (including physical ones) for the datacomputer than for the systems it serves.

2.2.2 Mass Storage Hardware

The datacomputer will store most of its data on mass storage devices, which have distinctive access characteristics. Two examples of such hardware are Precision Instruments' Unicon 690 and Ampex Corporation's TBM system. They are quite different from disks, and differ significantly from one another.

However, almost all users will be ignorant of the characteristics of these devices; many will not even know that the data they use is at the datacomputer. Finally, as the development of the system progresses, data may be invisibly shunted from one datacomputer to another, and as a result be stored in a physical format quite different from that originally used.

In such an environment, it is clear that requests for data should be stated in logical, not physical terms.

2.3 Network Environment

The network environment provides additional requirements for datacomputer design.

2.3.1 Remote Use

Since the datacomputer is to be accessed remotely, the requirement for effective data selection techniques and good mechanisms for the expression of selection criteria is amplified. This is because of the narrow path through which network users communicate with the datacomputer. Presently, a typical process-to-process transfer rate over the Arpanet is 30 kilobits per second. While this can be increased through optimization of software and protocols, and through additional expenditure for hardware and communications lines, it seems safe to assume that it will not soon approach local transfer rates (measured in the megabits per second).

A typical request calls for either transfer of part of a file to a remote site, or for selective update to a file already stored at the datacomputer. In both of these situations, good mechanisms for specifying the parts of the data to be transmitted or changed will reduce the amount of data ordinarily

transferred. This is extremely important because with the low per bit cost
of storing data at the datacomputer, transmission costs will be a significant
part of the total cost of datacomputer usage.

2.3.2 Interprocess Use of the Datacomputer System

Effective use of the network requires that groups of processes, remote from
one another, be capable of cooperating to accomplish a given task or provide
a given service. For example, to solve a given problem which involves array
manipulation, data retrieval, interaction with a user at a terminal, and the
generalized services of a language like PL/I, it may be most economical to
have four cooperating processes. One of these could execute at the ILLIAC
IV, one at the datacomputer, one at MULTICS, and one at a TIP.   While there
is   overhead in setting up these four processes and in having them
communicate, each is doing its job on a system specialized for that job.   In
many cases, the result of using the specialized system is a gain of several
orders of magnitude in economy or efficiency (for example, online storage at
the datacomputer has a capital cost two orders of magnitude lower than online
costs on conventional systems). As a result, there is considerable incentive
to consider solutions involving cooperating processes on specialized systems.

To summarize: the datacomputer must be prepared to function as a component of
small networks of specialized processes, in order that it can be used
effectively in a network in which there are many specialized nodes.

2.3.3 Common Network Data Handling

A large network can support enough data management hardware to construct more
than one datacomputer.   While this hardware can be combined into one even
larger datacomputer, there are advantages to configuring it as two (or
possibly more) systems.   Each system should be large enough to obtain
economies of scale in data storage and to support the data management
software.   Important data bases can be duplicated, with a copy at each
datacomputer; if one datacomputer fails, or is cut off by network failure,
the data is still available. Even if duplicating the file is not warranted,
the description can be kept at the different datacomputers so   that
applications which need to store data constantly can be guaranteed that at
least one datacomputer is available to receive input.

These kinds of failure protection involve cooperation between a pair of
datacomputers; in some sense, they require that the two datacomputers
function as a single system. Given a system of datacomputers (which one can
think of as a small network of datacomputers), it is obviously possible to
experiment with providing additional services on the datacomputer-network
level.   For example, all requests could be addressed simply to the
datacomputer-network; the datacomputer-network could then determine where
each referenced file was stored (i.e., which datacomputer), and how best to
satisfy the request.

Here, two kinds of cooperation in the network environment have been mentioned: cooperation among processes to solve a given problem, and cooperation among datacomputers to provide global optimizations in the network-level data handling problem. These are only two examples, especially interesting because they can be implemented in the near term. In the network, much more general kinds of cooperation are possible, if a little farther in the future. For example, eventually, one might want the datacomputer(s) to be part of a network-wide data management system, in which data, directories, services, and hardware were generally distributed about the network. The entire system could function as a whole under the right circumstances. Most requests would use the data and services of only a few nodes. Within this network-wide system, there would be more than one data management system, but all systems would be interfaced through a common language. Because the datacomputers represent the largest data management resource in the network, they would certainly play an important role in any network-wide system. The language of the datacomputer (datalanguage) is certainly a convenient choice for the common language of such a system.

Thus a final, albeit futuristic, requirement imposed by the network on the design of the datacomputer system, is that it be a suitable major component for network-wide data management systems. If feasible, one would like datalanguage to be a suitable candidate for the common language of a network-wide group of cooperating data management systems.

2.4 Different Modes of Datacomputer Usage

Within this network environment, the datacomputer will play several roles. In this section four such roles are described. Each of them imposes constraints on the design of datalanguage. We can analyze them in terms of four overlapping advantages which the datacomputer provides:

1. Generalized data mangement services
2. Large file handling
3. Shared access
4. Economic volume storage

Of course, the primary reason for using the datacomputer will be the data management services which it provides. However, for some applications size will be the dominating factor in that the datacomputer will provide for online access to files which are so large that previously only offline storage and processing were possible. The ability to share data between different network sites with widely different hardware is another feature provided only by the datacomputer. Economies of scale make the datacomputer a viable substitute for tapes in such applications as operating system backup.

Naturally, a combination of the above factors will be at work in most datacomputer applications. The following subsections describe some possible modes of interaction with the datacomputer.

2.4.1 Support of Large Shared Databases

This is the most significant application of the datacomputer, in nearly every sense.

Projects are already underway which will put databases of over one hundred billion bits online on the Arpanet datacomputer. Among these are a database which will ultimately include 10 years of weather observations from 5000 weather stations located all over the world. As online databases, these are unprecedented in size. They will be of international interest and be shared by users operating on a wide variety of hardware and in a wide variety of languages.

Because these databases are online in an international network, and because they are expected to be of considerable interest to researchers in the related fields, it seems obvious that there will be extremely broad patterns of use. A strong requirement, then, is a flexible and general approach to handling them. This requirement of providing different users of a database with different views of the data is an overriding concern of the datalanguage design effort. It is discussed separately in Section 2.5.

2.4.2 Extensions of Local Data management Systems

We imagine local data handling systems (data management systems, applications-oriented packages, text-handling systems, etc.) wanting to take advantage of the datacomputer. They may do so because of the economics of storage, because of the data management services, or because they want to take advantage of data already stored at the datacomputer. In any case, such systems have some distinctive properties as datacomputer users: (1) most would use local data as well as datacomputer data, (2) many would be concerned with the translation of local requests into datalanguage.

For example, a system which does simple data retrieval and statistical analysis for non-programming social scientists might want to use a census database stored at the datacomputer. Such a system may perform a range of data retrieval functions, and may need sophistocated interaction with the datacomputer. Its usage patterns would make quite a contrast with those of a single application program whose sole use of the datacomputer involves printing a specific report based on a single known file.

This social-science system would also use some local databases, which it keeps at its own site because they are small and more efficiently accessed locally. One would like it to be convenient to think of data the same way, whether it is stored locally or at the datacomputer. Certainly at the lower levels of the local software, there will have to be differences in interfacing; it would be nice, however, if local concepts and operations could easily be translated into datalanguage.

2.4.3 File Level Use of the Datacomputer

In this mode of use, other computer systems take advantage of the online
storage capacity of the datacomputer. To these systems, datacomputer storage
represents a new class of storage: cheaper and safer than tape, nearly as
accessible as local disk. Perhaps they even automatically move files between
local online storage and the datacomputer, giving users the impression that
everything is stored locally online.

The distinctive feature of this mode of use is that the operations are on
whole files.

A system operating in this mode uses only the ability to store, retrieve,
append, rename, do directory listings and the like. An obvious way to make
such file level handling easily available to the network community is to make
use of the File Transfer Protocol (see Network Information Center document
#17759 --File Transfer Protocol) already in use for host to host file
transfer.

Although such "whole file" usage of the datacomputer would be motivated
primarily by economic advantages of scale, data sharing at the file level
could also be a concern. For example, the source files of common network
software might reside at the datacomputer. These files have little or no
structure, but their common use dictates that they be available in a common,
always accessable place. It is taking advantage of the economics of the
datacomputer, more than anything else, since most of these services are
available on any file system.

This mode of use is mentioned here because it may account for a large
percentage of datalanguage requests. It requires only capabilities which
would be present in datalanguage in any case; the only special requirement is
to make sure it is easy and simple to accomplish these tasks.

2.4.4 Use of Datacomputer for File Archiving

This is another economics-oriented application. The basic idea is to store
on the datacomputer everything that you intend to read rarely, if ever. This
could include backup files, audit trails, and the like.

An interesting idea related to archiving is incremental archiving. A typical
practice, with regard to backing up data stored online in a time-sharing
system, is to write out all the pages which are different than they were in
the last dump. It is then possible to recover by restoring the last full
dump, and then restoring all incremental dumps up to the version desired.
This system offers a lower cost for dumping and storage, and a higher cost
for recovery; it is appropriate when the probability of needing a recovery is
low. Datalanguage, then, should be designed to permit convenient incremental
archiving.

As in the case of the previous application (file system), archiving is
important as a design consideration because of its expected frequency and
economics, not because it necessarily requires any extra generality at the
language level. It may dictate that specialized mechanisms for archiving be
built into the system.

2.5 Data Sharing

Controlled sharing of data is a central concern of the project. Three major sub-problems in data sharing are: (1) concurrent use, (2) independent concepts of the same database, and (3) varying representations of the same database.

Concurrent use of a resource by multiple independent processes is commonly implemented for data on the file level in systems in which files are regarded as disjoint, unrelated objects. It is sometimes implemented on the page level.

Considerable work on this problem has already been done within the datacomputer project. When this work is complete, it will have some impact on the language design; by and large however, we do not consider this aspect of concurrent use to be a language problem.

Other aspects of the concurrent use problem, however, may require more conscious participation by the user. They relate to the semantics of collections of data objects, when such collections span the boundaries of files known to the internal operating system. Here the question of what constitutes an update conflict is more complex. Related questions arise in backup and recovery. If two files are related, then perhaps it is meaningless to recover an earlier state of one without recovering the corresponding state of the other. These problems are yet to be investigated.

Another problem in data sharing is that not all users of a database should have the same concept of that database. Examples: (1) for privacy reasons, some users should be aware of only part of the database (e.g., scientists doing statistical studies on medical files do not need access to name and address), (2) for program-data independence, payroll programs should access only data of concern in writing paychecks, even though skill inventories may be stored in the same database, (3) for global control of efficiency, simplicity in application programming, and program-data independence, each application program should "see" a data organization that is best for its job.

To further analyze example (3), consider a database which contains information about students, teachers, subjects and also indicates which students have which teachers for which subjects. Depending on the problem to be solved, an application program may have a strong requirement for one of the following organizations:

(1) entries of the form (student,teacher,subject) with no concern about redundancy. In this organization an object of any of the three types may occur many times.

(2) entries of the form

       (student,   (teacher,subject),
                    (teacher,subject',

                   .
                   .
                   .

```
                                    (teacher,subject))
(3) entries of the form
        (teacher,          subject,(student...student),
                           subject,(student...student),
                           subject,(student...student))
```
and other organizations are certainly possible.

One approach to this problem is to choose an organization for stored data,
and then have application programs write requests which organize output in
the form they want. The application programmer applies his ingenuity in
stating the request so that the process of reorganization is combined with
the process of retrieval, and the result is relatively efficient. There are
important, practical situations in which this approach is adequate; in fact
there are situations in which it is desirable. In particular, if efficiency
or cost is an overriding consideration, it may be necessary for every
application programmer to be aware of all the data access and organization
factors. This may be the case for a massive file, in which each retrieval
must be tuned to the access strategy and organization; any other mode of
operation would result in unacceptable costs or response times.

However, dependence between application programs and data organization or
access strategy is not a good policy in general. In a widely-shared
database, it can mean enormous cost in the event of database reorganization,
changes to access software, or even changes in the storage medium. Such a
change may require reprogramming in hundreds of application programs
distributed throughout the network.

As a result, we see a need for a language which supports a spectrum of
operating modes, including: (1) application program is completely independent
of storage structure, access technique, and reorganization strategy, (2)
application program parametrically controls these, (3) application program
entirely controls them. For a widely-shared database, mode (1) would be the
preferred policy, except when (a) the application programmer could do a
better job than the system in making decisions, and (b) the need for this
increment of efficiency outweighed the benefits of program-data independence.

In evaluating this question for a particular application, it is important to
realize the role of global efficiency analysis. When there are many users of
a database, in some sense the best mode of operation is that which minimizes
the total cost of processing all requests and the total cost of storing the
data. When applications come and go, as real-world needs change, then the
advantages of centralized control are more likely to outweigh the advantages
of optimization for a particular application program.

The third major sub-problem arises in connection with item level
representations. Because of the environment in which it executes, each
application program has a preferred set of formatting concepts, length
indicators, padding and alignment conventions, word sizes, character
representations, and so on. Once again it is better policy for the
application program to be concerned only with the representations it wants
and not with the stored data representation. However, there will be cases in
which efficiency for a given request overrides all other factors.

At this level of representation, there is at least one additional
consideration: potential loss of information when conversion takes place.
Whoever initiates a type conversion (and this will sometimes be the
datacomputer and sometimes the application program) must also be responsible
for seeing that the intent of the request is preserved. Since the
datacomputer must always be responsible for the consistency and the meaning
of a shared database, there are some conflicts to be resolved here.

To summarize, it seems that the result of wide sharing of databases is that a
larger system must be considered in choosing a data management policy for a
particular database. This larger system, in the case of the datacomputer,
consists of a network of geographically distributed applications programs, a
centralized database, and a centralized data management system. The
requirement for datalanguage is to provide flexibility in the management of
this larger system. In particular, it must be possible to control when and
where conversions, data re-organizations, and access strategies are made.

2.6 Need for High Level Communication

All of the above considerations point to the need for high level
communication between the datacomputer and its users. The complex and
distinct nature of datacomputer hardware make it imperative that requests be
put to the datacomputer so that it can make major decisions regarding the
access strategies to be used. At the same time, the large amounts of data
stored and the demand of some users for extremely high transmission
bandwidths make it necessary to provide for user control of some storage and
transmission schemes. The fact that databases will be used by applications
which desire different views of the same data and with different constraints
means that the datacomputer must be capable of mapping one users request onto
another users data. Interprocess use of the datacomputer means that
datasharing must be completely controllable to avoid the need for human
intervention. Extensive facilities for ensuring data integrity and
contolling access must be provided.

2.6.1 Data Description

Basic to all these needs is the requirement that the data stored at the
datacomputer be completely described in both functional and physical
parameters. A high level description of the data is especially important to
provide the sharing and control of data. The datacomputer must be able to
map between different hardware and different applications. In its most
trivial form this means being able to convert between floating point number
represenatations on different machines. On the other extreme it means being
able to provide matrix data for the ILLIAC IV as well as being able to
provide answers to queries from a natural language program, both addressed to
the same weather data base. Data descriptions must provide the ability to
specify the bit level representations and the logical properties and
relationships of data.

2.6.2 Data Integrity and Access Control

In the environment we have been describing, the problems of maintaining data
integrity and controlling use of data assume extreme importance. Shared use
of datacomputer files depends on the ability of the datacomputer to guarantee
that the restrictions on data access are strictly enforced. Since different
users will have different descriptions, the access control mechanism must be
associatied with the descriptions themselves. One can control access to data
by controlling access to its various descriptors. A user can be constrained
to access a given data base only through one specific description which
limits the data he can access. In a system where the updaters of a database
may be unknown to each other, and possibly have different views of the data,
only the datacomputer can assure data integrity. For this reason, all
restrictions on possible values of data objects, and on possible or
neccessary relationships between objects must be stated in the data
description.

2.6.3 Optimization

The decisions regarding data access strategy must ordinarily be made at the
datacomputer, where knowledge of the physical considerations is available.
These decisions cannot be made intelligently unless the requests for data
access are made at a high level.

For example, compare the following two situations: (1) a request calls for
output of _all_ weather observations made in California exhibiting certain wind
and pressure conditions, (2) a series of requests is sent, each one
retrieving California weather observations; when a request finds an
observation with the required wind and pressure conditions, it transmits this
observation to a remote system. Both sessions achieve the same result: the
transmission of a certain set of observations to a remote site for
processing. In the first session, however, the datacomputer receives, at the
outset, a description of the data that is needed; in the second, it processes
a series of requests, each one of which is a surprise.

In the first case, a smart datacomputer has the option of retrieving all of
the needed data in one access to the mass storage device. It can then buffer
this data on disk until the user is ready to accept it. In the second case,
the datacomputer lacks the information it needs to make such an optimization.

The language should permit and encourage users to provide the information
needed to do optimization. The cost of not doing it is much higher with mass
storage devices and large files than it is in conventional systems.

2.7 Application Oriented Concerns

In the above sectons we have described a number of features which the
datacomputer system must provide. In this section we focus on what is
necessary to make these features readily available to users of the

datacomputer.

2.7.1 Datacomputer-user Interaction

An application interacts with the datacomputer in a <u>session</u>. A session
consists of a series of <u>requests</u>. Each session involves connecting to the
datacomputer via the network, establishing identities, and setting up
transmission paths for both data and datalanguage. Datalanguage is
transmitted in character mode (using network standard ASCII) over the
datalanguage connection. Error and status messages are sent over this
connection to the application program.

The data connection (called a PORT) is viewed as a bit stream and is given
its own description. These descriptions are similar to those given for
stored data. At a minimum this description must contain enough information
for the datacomputer to parse the incoming bit stream. It also may contain
data validation information as well. To store data at the datacomputer, the
stored data must also have a description. The user supplies the mapping
between the descriptions of the stored and transmitted data.
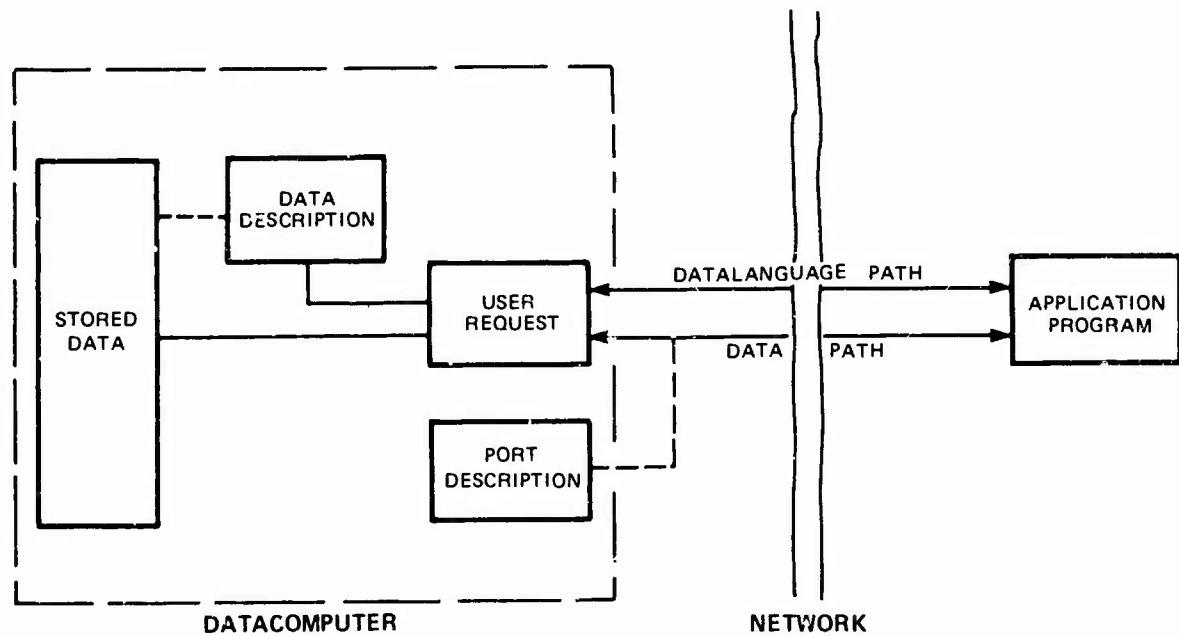
Figure 2-1
A Model of Datacomputer/User Interaction

2.7.2 Application Features for Data Sharing

In using data stored at the datacomputer, users may supply a description of
the data which is customized to the application. This description is mapped
onto the description of the stored data.   These descriptions may be at
different levels.   That is, one may merely rearrange the order of certain
items, while another could call for a total restructuring of the stored
representation. So that each user may be able to build upon the descriptions
of another, data entities should be given named types. These type
definitions are of course to be stored along with the data they describe. In
addition, certain functions are so closely tied to the data (in fact may be
the data in the virtual desciption case -- see section 3), that they must
also reside in the datacomputer and their tie with the data items should be
maintained by the datacomputer. For example, one user can describe a data
base as made up of structures containing data of the types latitude and
longitude. He could also describe functions for comparing data of this type.
Other users, not concerned with the structure of the latitude component
itself, but interested in using this information simply to extract other
fields of interest can then use the commonly provided definitions and
functions.   Futhermore, by adopting this strategy as many users as possible
can be made insensitive to changes in the file which are tangential to their
main interests.   For example, latitudes could be changed from binary
representation to a character form and if use of that field were restricted
to its definitions and associated functions, existing application systems
would be unaffected. Conversion functions could be defined to eliminate the
impact on currently operating programs. The ability of such definitional
facilities means that groups of users can develop common functions and
descriptions for dealing with shared data and that conventions for use of
shared data can be enforced by the datacomputer.   These facilities are
discussed under extensibility in Section 3.
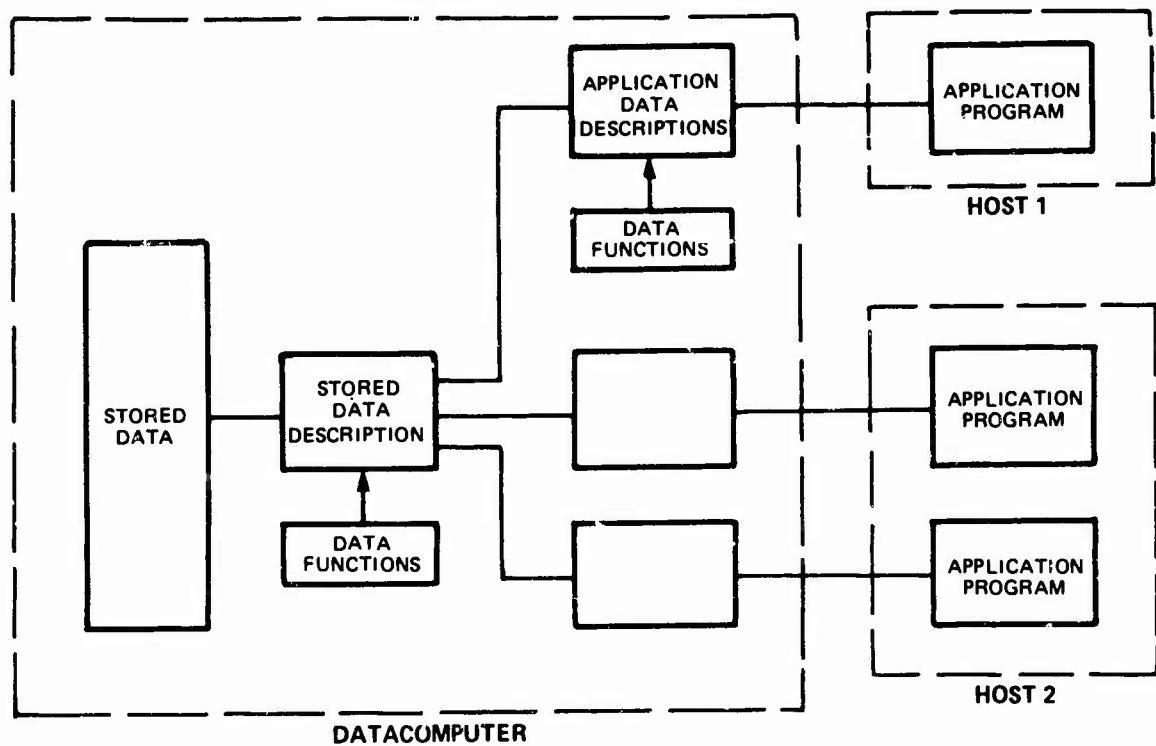
Figure 2-2
Multiple User Interaction with the Datacomputer

2.7.3 Communication Model

We intend that datalanguage, while at a high level conceptually, will be at a
low level syntactically. Datalanguage provides a set of primitive functions,
and a set of commonly used higher level functions (see section 4 on the
datalanguage model). In addition, users can define their own functions so
that they can communicate with the datacomputer at a level as conceptually
close to the application as possible.

There are two reasons for datalanguage being at a low level syntactically.
First, it is undesirable to have programs composing requests into an
elaborate format only to be decomposed by the datacomputer. Second, by
choosing a specific high level syntax, the datacomputer would be imposing a
set of conventions and terminology which would not necessarily correspond to
those of most users.

DATACOMPUTER ENVIRONMENT   OUTSIDE ENVIRONMENT

```
  ┌──────────┐   ┌──────────┐   ┌──────────┐          ┌──────────┐
  │PRIMITIVE │   │HIGHER    │   │          │          │ GENERAL  │
  │LANGUAGE  │───│LEVEL     │───│LOW-LEVEL │──────────│   DMS    │──
  │CONCEPTS  │   │LANGUAGE  │   │ SYNTAX   │          │          │
  │          │   │CONCEPTS  │   │          │          └──────────┘
  └──────────┘   └──────────┘   └──────────┘
                                              ┌──────────┐   ┌──────────┐
                                              │  COBOL   │   │  COBOL   │
                                              │ SERVER   │───│ PROGRAM  │
                                              └──────────┘   └──────────┘

                                              ┌──────────┐
                                              │ ON LINE  │
                                              │  QUERY   │───┐
                                              │ SYSTEM   │   │  ┌──────────┐
                                              └──────────┘   │  │ TERMINAL │
                                                             └──│  USERS   │
                                                                └──────────┘
```

APPLICATION   APPLICATIONS
SERVERS
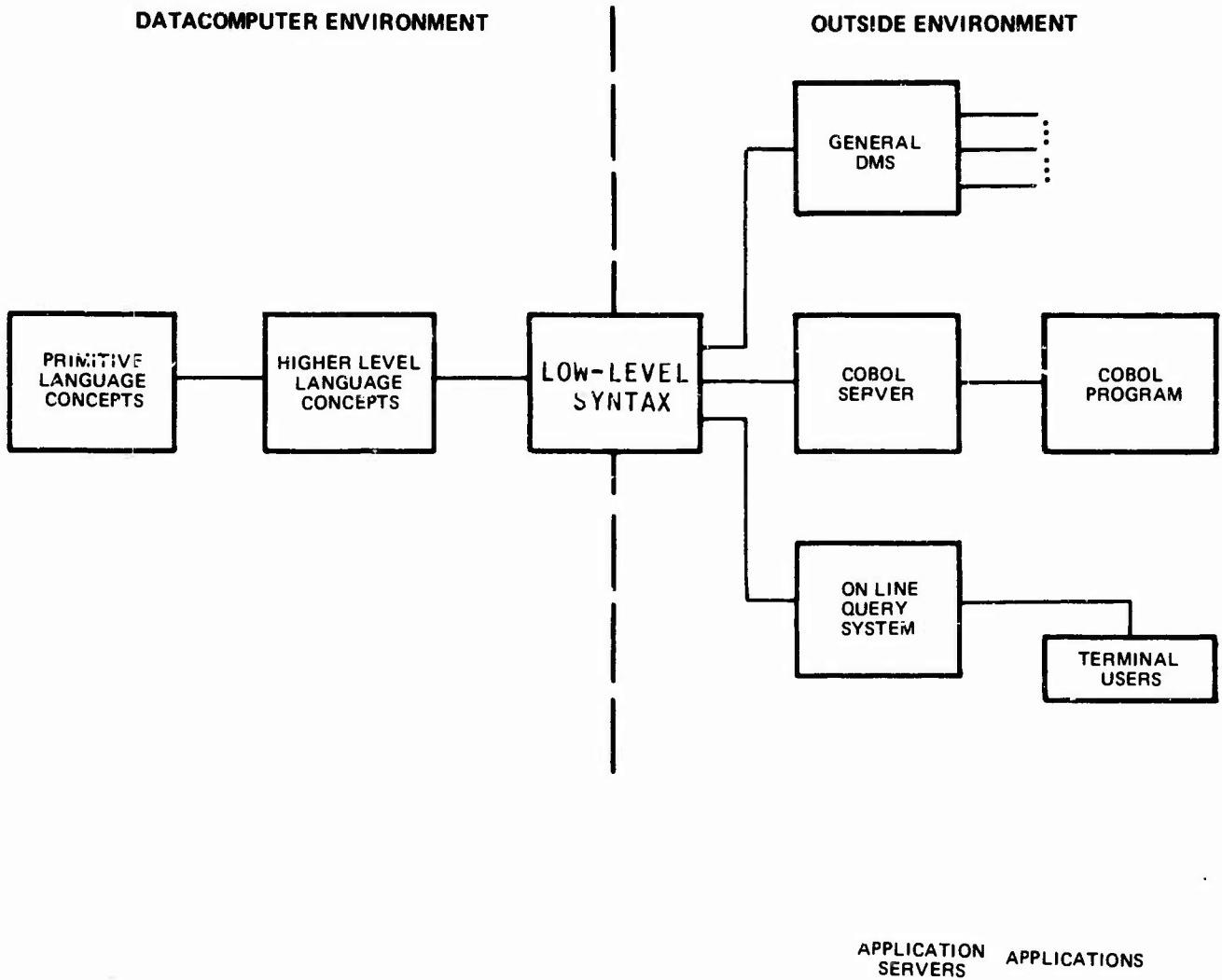
Figure 2-3
Datacomputer/User Working Environment

2.8 Summary

In this section we have presented the major considerations which have
influenced the current datalanguage design effort. The datacomputer has much
in common with most large-scale shared data management systems, but also has
a number of overriding concerns unique to the datacomputer concept. The most
important of these are the existence of a separate box containing both
hardware and software, the control of an extremely large storage device, and
imbedding in a computer network environment. Data sharing in such an
environment is a central concern of the design.   Both extensive data

description facilities and high level communication between user and datacomputer are necessary for data integrity and for datacomputer optimization of user requests. In addition, the expected use of the datacomputer involves satisfying several conflicting contraints for different modes of operation. One way of satisfying various user needs is to provide datalanguage features so that users may develop their own application packages within datalanguage.

3. Principal Language Concepts

This section discusses the principal facilities of datalanguage.   specific
details of the language are not presented, however, the discussion includes
the motivation behind the inclusion of the various language features and also
defines, in an informal way, the terms we use.

3.1 Baric Data Items

Basic data are the atomic level of all data constructions.   they cannot be
decomposed.   All higher level data structures are fundamentally composed of
basic data items. Many types of basic data items will be provided. The type
of an item determines what operations can be performed on the item and the
meaning of those operations. Datalanguage will provide those primitive types
of data items which are commonly used in computing systems to model the real
world.

The following basic types of data will be available in datalanguage: fixed
point numbers, floating point numbers, characters, booleans, and bits. These
types of items are "understood" by the datacomputer system to the extent that
operations are based on the type of an item. Datalanguage will also include
an uninterpreted type of item, for data which will only be moved (including
transmitted) from one place to another.   This type of data will only be
understood in the trivial sense that the datacomputer can determine if two
items of the uninterpreted type are identical. Standard operations on the
basic types of items will be available. Operations will be included so that
the datacomputer user can describe a wide range of data management functions.
They are not included with the intent of encouraging use of the datacomputer
for the solving of highly computational problems.

3.2 Data Aggregates

Data aggregates are compositions of basic data items and possibly other data
aggregates.   The types of data aggregates which are provided allow for the
construction of hierarchical relationships of data.   The aggregates which
will definitely be available are classified as structs, arrays, strings,
lists, and directories.

A struct is a static aggregate of data items (called components).   A struct
is static in the sense that the components of a struct cannot be added or
deleted from the struct, they are inextricably bound to the struct.
Associated with each component of the struct is a name by which that
component may be referenced relative to the struct. The struct aggregate may
be used to model what is often thought of as a record, with each component
being a field of that record. A struct can also be used to group components
of a record which are more strongly related, conceptually, than other
components and may be operated on together.

Arrays allow for repitition in data structures. An array, like a struct, is
a static aggregate of data items (called members). Each member of an array
is of the same type. Associated with each member is an index by which that
member can be referenced relative to the array. Arrays can be used to model
repeating data in a record (repeating groups).

The concept of string is actually a hybrid of basic data and data aggregates.
Strings are aggregates in that they are compositions (similar to arrays) of
more primitive data (e.g., characters).   They are, however, generally
conceived of as basic in that they are mostly viewed as a unit rather than as
a collection of items, where each item has individual importance.   Also the
meaning of a string is highly dependent on the order of the individual
components.   In more concrete terms, there are operations which are defined
on specific types of strings. For example, the logical operators (and, or,
etc.) are defined to operate on strings of bits.   However, there are no
operations which are defined on arrays of bits, although there are operations
defined on both arrays, in general, and on bits.   Strings of characters,
bits, and uninterpretted data will be available in datalanguage.

Lists are like arrays in that they are collection of similar members.
However, lists are dynamic rather than static. Members of a list can be
added and deleted from the list. Although, the members of a list are ordered
(in fact more than one ordering can be defined on a list), the list is not
intended to be referenced via an index, as is the case with an array.
Members of a list can be referenced via some method of sequencing through the
list. A list member, or set (see discussion under virtual data) of members,
can also be referenced, by some method of identification by content. The
list structure can be used to model the common notion of a file.   Also
restrictive use of lists as components of structs provides power with respect
to the construction of dynamic hierarchical data relationships below the file
level.   For example, the members of a list may themselves be, in part,
composed of lists, as in a list of families, where each family contains a
list of children as well as other information.

Directories are dynamic data aggregates which may contain any type of data
item. Data items contained in a directory are called nodes. Associated with
each node of a directory is a name by which that data item can be referenced
relative to the directory. As with lists, items may be dynamically added to
and deleted from a directory. The primary motivation behind providing the
directory capability is to allow the user to group conceptually related data
together. Since directories need not contain only file type information,
"auxilliary" data can be kept as part of the directory. For example,
"constant" information, like salary range tables for a corporation data base;
or user defined operations and data types (see below) can be maintained in a
directory along with the data which may use this information.   Also
directories may themselves be part of a directory, allowing for a hierarchy
of data grouping.

Directories will also be defined so that system controlled information can be
maintained with some of the subordinate items (eg. time of creation, time of
update, privacy locks, etc.). It may also be possible to allow the data user
to define and control his own information which would be maintained with the
data. At the least, the design of datalanguage will allow for parametric

control over the information managed by the system.

Directories are the most general and dynamic type of aggregate data. Both
the name and description (see below) of directory nodes exist with the nodes
themselves, rather than as part of the description of the directory. Also
the level of nesting of a directory is dynamic since directories can be
dynamically added to directories. Directories are the only aggregate for
which this is true.

Datalanguage will also provide some specific and useful variations of the
above data aggregates. Structs will be available which allow for optional
components. In this case the existence of a component would be based on the
contents of other components. It may also be possible to allow for the
existence to be based on information found at a higher level of data
hierarchy. Similarly, components with unresolved type will be provided.
That is the component may be one of a fixed number of types. The type of the
component would be based on the contents of other components of the struct.
It is also desirable to allow the type or existence of a component to be
based on information other than the contents of other components. For
instance, the type of one component might be based on the type of another
component. In general, we would like for datalanguage to allow for the
attributes (see below) of one item to be a function of the attributes of
other items.

We would also like to provide mixed lists. Mixed lists are lists which
contain more than one type of member. In this case the members would have to
be self defining. That is, the type of all member would have to be "alike"
to the degree that information which defines the type of that member could be
found.

Similar to components whose type is unresolved are Arrays with unresolved
length. In this case, information defining the length of the array must be
carried with the array or perhaps with other components of an aggregate which
encompasses the array.

In all of the above cases the type of an item is unresolved to some degree
and information which totally resolves the type is carried with the item. It
is possible that in some or perhaps all of these cases the datacomputer
system could be responsible for the maintainance of this information, making
it invisible to the data user.

3.3 General Relational Capabilities

The data aggregates described above allow for the modeling of various
relationships among data. All relationships which can be constructed are
hierarchical.

Two approaches can be taken to provide the capability of modeling
non-hierarchical relationships. New types of data aggregates can be
introduced which will braoden the range of data relationships expressable in
datalanguage. Or, a basic data type of "pointer" can be introduced which

will serve as a primitive out of which relations can be represented. Pointer would be a data type which establishes some kind of correspondence from one item to another. That is, it would be a method of finding one item, given another. Providing the ability to have items of type pointer does not necessitate the introduction of the concept of address which we deem to be a dangerous step. For example, an item defined to point to a record in a personnel file could contain a social security number which is contained in each record of the file and uniquely identifies that record. In general a pointer is an item of information which can be used to uniquely identify another item.

While the pointer approach provides the greater degree of flexibility, it does this at the price of relegating much of the work to the user as well as severely limiting the amount of control the datacomputer system has over the data. A hybrid solution is possible, where some new aggregate data types are provided as well as a restricted form of pointer data type. While the approach to be taken is still being studied, the datalanguage design will include some method of expressing non-hierarchical data structures.

3.4 Ordering of Data

Lists are generally viewed as ordered. It is possible, however, that a list can be used to model a dynamic collection of similar items which are not seen as ordered. The unordered case is important, in that, given this information the datacomputer can be more efficient since new members can be added wherever it is convenient.

There are a number of ways a list can be ordered. For instance, the ordering of a list can be based on the contents of its members. In the simplest case this involves the contents of a basic data item. For example, a list of structs containing information on employees of a company may be ordered on the component which contains the employee's social security number. More complex ordering criteria are possible. For example, the same list could be ordered alphabetically with respect to the employee's last name. In this case the ordering relation is a function of two items, the last and first names. The user might also want to define his own ordering scheme, even for orderings based on basic data items. An ordering could be based on an employee's job title which might even utilize auxilliary data (i.e. data external to the list). It is also possible to maintain a list in order of insertion. In the most general case, the user could dynamically define his ordering by specification of where an item is to be placed as part of his insertion requests. In all of the above cases, data could be maintained in ascending or descending order.

In addition to maintainance of a list in some order, it is possible to define one or more orderings "imposed" on a list. These orderings must be based on the contents of a list's members. This situation is similar to the concept of virtual data (see below) in that the list is not physically maintained in a given order, but retrieved as if it were. Orderings of this type can be dynamically formed (see discussion of set under virtual data). imposed orderings can be accomplished via the maintainance of auxilliary structures (see discussion under internal representation) or by utilization of a sorting strategy on retrievals. Much work has been done with regard to effective implementation of the maintainance and imposition of orderings on lists . This work is described in working paper number 2.

3.5 Data Integrity

An important feature of any data management system is the ability to have the system insure the integrity of the data. Data needs to be protected against erroneous manipulation by people and against system failure.

Datalanguage will provide automatic validity checks. Many flavors need to be provided so that appropriate trade-offs can be made between the degree of insurance and the cost of validation. The datalanguage user will be able to request constant validation: where validity checks are made whenever the data is updated; validation on access: where validity checks are performed when data is referenced but before it is retrieved. regularly scheduled validation: where the data is checked at regular intervals; background

validation: where the system will run checks in its spare time; and
validation on demand. constant validation and validation on access are
actually special cases of the more general concept of event triggered
validation. In this case the user specifies an event which will cause data
validation procedures to be invoked. This feature can be used to accomplish
such things as validation following a "batch" of updates. Also, some
mechanism for specifying combinations of these types would be useful.

In order for some of the data validation techniques to be effective, it may
be necessary to keep some data validation "bookkeeping" information with the
data. For example, information which can be used to determine whether an
item has been checked since it was last updated might be used to cause
validation on access if there has not been a recent background validation.
The datacomputer may provide for optional automatic maintainance of such
special kinds of information.

In order for the datacomputer system to insure data validity, the user must
define what valid is. Two types of validation can be requested. In the
first case the user can tell the datacomputer that a specific data item may
only assume one of a specific set of values. For example, the color
component of a struct may only assume the values 'red', 'green', or 'blue'.
The other case is where some relation must hold between members of an
aggregate. For example, if the sex component of a struct is 'male' then the
number of pregnancies component must be 0.

Data validation is only half of the data integrity picture. Data integrity
involves methods of restoring damaged data. This requires maintenance of
redundant information. Features will be provided which will make the
datacomputer system responsible for the maintainance of redundant data and
possibly even automatic restoration of damaged data. In section 2 we
discussed possible uses of the datacomputer for file backup. All features
which are provided for this purpose will also be available as methods of
maintaining backup information for restoration of files residing at the
datacomputer.

3.6 Privacy

Datalanguage will have to provide extensive privacy and protection
capabilities. In its simplest form a privacy lock is provided at the file
level. The lock is opened with a password key. Associated with this key is
a set of priveleges (reading, updating, etc.). Two degrees of generality are
sought. Privacy should be available at all levels of data. Therefore,
groups of related data, including groups of files could be made private by
creating private directories. also, specific fields of records could be made
private by having private components of a struct where other components of
the struct are visible to a wider (or different) class of users. We would
also like the user to be able to define his own mechanism. In this way, very
personalized, complex, and hence secure mechanisms can be defined. Also
features such as 'everyone can see his own salary' might be possible.

3.7 Conversion

Many types of data are related in that some or all of the possible values of
one type of data have an "obvious" translation to the values of another. For
example, the character '6' has a natural translation to the integer 6, or the
six character string 'abc ' (three trailing blanks) has a natural
translation to the four character string 'abc ' (one trailing blank).
Datalanguage will provide conversion capabilities for the standard, commonly
called for, translations. These conversions can be explicitly invoked by the
user or implictly invoked when data of one type is needed for an operation
but data of another type is provided. In the case of implicit invocation of
conversion of data the user will have control over whether conversion takes
place for a given data item. More generally we would like to provide a
facility whereby the user could specify conditions which determine when an
item is to be converted. Also, the user should be able to define his own
conversion operations, either for a conversion between types which is not
provided by the datacomputer system or to override the standard conversion
operation for some or all items of a given type.

3.8 Virtual and Derived Data

Often, information important to users of data is imbedded in that data rather
than explicitly maintained. For example, the dollar value of an individual's
interest in a company in a file of stock holders. Since the value of the
company changes frequently, it is not feasible to maintain this information
with each record. It is useful to be able to use the file as if information
of this type was part of each record. When referencing the dollar value
field of a record, the datacomputer system would automatically use
information in the record, such as percentage of ownership in the company,
possibly in conjuction with information which is not part of the record but
is maintained elsewhere, such as company assets, to compute the dollar value.
In this way the data user need not be concerned with the fact that this
information is not actually maintained in the record.

The set, which is a specific type of virtual container in datalanguage,
deserves special mention.   A set is a virtual list. For example, suppose
there is a real list of people representing some population sample. By real
(or actual) data we mean data which is physically stored at the datacomputer.
A set could be defined to contain all members of this list who are automobile
owners.   The set concept provides a powerful feature for viewing data as
belonging to more than one collection without physical duplication. Sets are
also useful, in that, they can be dynamically formed. Given an actual list,
sets based on that list can be created without having been previously
described.

As mentioned above, virtual data can be very economical. These economies may
become most important with respect to the use of sets. Savings are found not
only in regard to storage requirements, but also in regard to processing
efficiency. Processing time can be reduced as a result of calculations being
performed only when the data is accessed. The ability to obtain efficient
operation by optimization becomes greater when virtual data is defined in
terms of other virtual data. For sets, large savings may be realized by

straight forward "optimization" of the nested calculations.

The above ideas are made more clear by example. Having created a set of automobile owners, A, a set of home owners, HA, can be defined based on A. The members of HA can be produced very efficiently, in one step, by retrieving people who are both automobile owners and home owners. This is more efficient than actually producing the set, A and then using it to create HA. This is true when one or both pieces of information (automobile ownership and home ownership) are indexed (see discussion under internal representation) as well as when neither is indexed.

The same gains are achieved when operations on virtual data are requested. For example, if a set, H, had been defined as the set of homeowners based on the original list of people, the set, HA, could have been defined as the intersection (see discussion on operators) of A and H. In this case too, HA can be calculated in one step. Use of sets allows the user to request data manipulations in a form close to his conceptual view, leaving the problem of effective processing of his request to the datacomputer.

Another use of virtual data is to accomplish data sharing. An item could be defined, virtually, as the contents of another item. If no restriction is placed on what this item can be, we have the ability to define two paths of access to the same data. Hence, data can be made subrodinate to two or more aggregate structures. Stated another way, there are two or more paths of access to the data. This capability can be used to model data which is part of more than one data relationship. For example, two files could have the same records without maintaining duplicate copies.

It will also be possible, via data sharing to look at data in different ways. Shared data might behave differently depending on how (and ultimately by whom) it is accessed. Although, the ability to nave multiple paths to the same data and the ability to have data which is calculated on access are both part of the general virtual data capability, datalanguage will probably provide these as separate features, since they have different usage characteristics.

Derived data is similar to virtual data in that it is redundant data which can be calculated from other information. Unlike virtual data it is physically maintained. The user can choose between virtual and derived data as a result of considering trade-offs based on: estimated cost of calculation; frequency of update; estimated cost of storage; and frequency of access. For example, suppose a file contains a list of budgets for various projects in a department. The departmental budget can be calculated as a function of the individual project budgets. This information might be defined as derived data since it is expected to be updated infrequently (e.g., once a year), while it is expected to be accessed relatively often.

Options will be provided which give the user control with regard to when the calculation of derived data is to be done. These options will be similar to those provided for control of data validity operations. The data validation and derived data concepts are similar in that some operation must be performed on related data. In the case of data validation, the information derived is the condition of data.

3.9 Internal Representation

To this point, we have discussed only the high level, logical, aspects of
data.   Since data, at any given time, must reside on some physical device a
representation of the data must be chosen. In some cases it is appropriate
to   leave this choice to the datacomputer system. For example, the
representation of information which is used in the process of transmiting
other data, but which itself resides solely at the datacomputer may not be of
any concern to the user.

However, it is important that the user be capable of controlling the choice
of representation. In any application which requires mostly transmission of
data rather than interpretation of the data by the datacomputer, the data
should be maintained in a form consistent with the system which communicates
with the datacomputer. With respect to basic types of data, datalanguage
will provide most representation commonly used in systems with which it
interacts. For some types (e.g., fixed point) this will be accomplished by
providing for parametric (e.g., sign convention, size) description of the
representation.   In   other   cases   (e.g.,   floating   point)   specific
representations will be offered (e.g., system 360 short floating point,
system 360 long floating point, pdp-10 floating point, etc.).

Another aspect of the internal representation problem regards aggregate
structures.   The method chosen to represent aggregate structures may largely
affect the cost of manipulating the data. The user must have control over
this representation since only he has any idea of how the data is to be used.
Datalanguage will provide a variety of representational options which will
allow for efficient implementation of data structures. This includes the
availability of auxilliary structures, automatically maintained by the data
computer system. These structures can be used to effect efficient retrieval
of subsets of data collections based on the contents of the members (i.e.
the common concept of indicies), efficient maintainance of orderings on a
collection of data, maintainance of redundant information for the purpose of
data integrity, and efficient handling of shared data whose behavioral
characteristics are dependent on the path of access. It should be noted here
that, the datalanguage design effort, will attempt to provide methods whereby
the data user can describe the expected use of his data, so that details of
internal representation can be left to the datacomputer.

3.10 Data Attributes and Data Classes

The type of an item determines the operations which are valid on that item
and what they mean.   data attributes are refinements on the type of data.
The data attributes affect the meaning of operations. For example, we would
like to provide for the option of defining fixed point items to be scaled.
The scale factor, in this case, would be an attribute of fixed point data.
It effects the meaning of operations on that data. The attribute concept is
useful in that it allows information concerning the manipulation of an item
to be associated with the item rather than with the invocation of all
operations on that item.

The attribute concept can be applied to aggregate as well as basic data. For example, one attribute of a list could define where a new member is to be inserted. Options might be: insert at the beginning of the list; insert at the end of the list; or insert in some order based on the contents of the member. Adding a new member to a list with one of the above attributes could be done by issuing a simple insert request without having to specify where the new member is to be inserted.

The data class concept is actually the inverse of the data attribute concept. A data class is a collection of data types. The data class concept allows for definition of operations, independent of specific type of an item. For example, by defining the data class arithmetic to be composed of fixed point and floating point types of data, the comparison operators (equal, less than, etc.) can be defined to operate on arithmetic data, independent of whether it is fixed or floating point. Also the concept of data aggregate can be seen as a class encompassing directories, lists, etc. As there are operations defined on arithmetic data, there are also operations defined on arbitrary aggregates.

The inverse relationship between data classes and data attributes is very strong. For example, the concept of list can be seen as a data class, encompassing all types of lists (e.g., lists of integers, lists of character strings, etc.), independent of the types of their members. The type of a list's members (e.g., integer, character string, etc.) are then seen as attributes. Data attributes and classes are also relative concepts. While the concept of list can be viewed as a data class, it can also be seen as an attribute, relative to the concept of data aggregate.

3.11 Data Description

A data description is a statement of the properties (see discussion of attributes) of a data item. Examples of properties which are recorded in a description are: the name of an item; its size; its data type; its internal representation; privacy information; etc.

Datalanguage will contain mechanisms for specifying data descriptions. These descriptions will be processed by the data computer, and used whenever the data item is referenced. The user will be able to physically create data only by first specifying their descriptions. The properties of a description can be divided into groups according to thier function. Some have the function of specifying details of representation, which will not be of interest to most users, while others, such as the name are of almost universal interest.

All user data is a part of some larger (user or system) data structure. The structures containing data establish a path of access to the data. In the process of following this path the datacomputer system must accrue a complete description of the data item. For example, the description of a data item of a directory may be found associated with that node of the directory. Members of a list or array are described as part of the description of the list or array. We must dispose of two seeming exceptions. First, while aspects of

data may (on user request) be left to the system, those aspects are still
described, they are described by the system. As discussed above, some data
will be, to some degree, self describing (eg. members of mixed lists).
However, it is fully described in some encompassing structure, in that a
method of determing the full description is described.

It is worth noting here that the sooner a complete description is found in
the path of access, the more effective the datacomputer is likely to be in
processing requests which manipulate a data item. However, the ability to
have data whose complete description does not exist at high levels of the
access path provides greater flexibility in the definition of data
structures.

3.12 Data Reference

Data cannot be manipulated unless it can be referenced. In the same way that
data cannot exist without its being described, it cannot exist unless there
is a path of access to the data. The method of data reference is to define
the path of access to the data. As mentioned above, there is a method of
referencing any item relative to the data aggregate which contains it. Nodes
of directories and components of structs are referenced via the name
associated with the node or component. Members of arrays are referenced via
the index associated with the member. Members of lists are referenced via
some method of specifying the position of the member or by uniquely
identifying the member by content. To reference any arbitrary data item the
path of access must be fully defined by either explicit or implicit
definition of each link in the chain. In the case of virtual data there is
an extra implicit link in the chain, that being the method employed to obtain
the data from other data items. It should be noted also that if pointers are
provided (see discussion on general relational capabilities) they can also
serve as a link in the chain of access to an item.

The design of datalanguage will ease the problem (and reduce the cost) of
referencing data items by providing methods whereby part of the access path
can be implicitly defined. For example, datalanguage will provide a concept
of "context". During the course of interacting with the datacomputer, levels
of context can be set up so that data can be referenced directly, in context.
for example, on initiating a session the user may (in fact will probably be
required) to define a directory which will be the context of that session.
All items subordinate to this directory can be referenced directly in this
context. Another feature will be partial qualification. Each level of
struct need not be mentioned in order to reference an item imbedded in a deep
nest of structs. Only those intermediate levels which are sufficient to
uniquely identify the item need be specified.

3.13 Operations

In this section we discuss the builtin functions of datalanguage which are of
central importance in manipulating data. Functions which operate on items,
functions which operate on aggregates, primitive functions and high-level
functions are discussed.

Of the primitives which operate on items, those of most interest are assignment, comparisons, logicals, arithmetics and conversion functions.

Primitive assignment transfers a value from one item to another; these items must be of the same type. When they are of different types, either conversion must be performed, or some non-primitive form of assignment is involved.

The comparison operators accept a pair of items of the same type, and return a boolean object which indicates whether or not a given condition obtains. The type determines how many different conditions can be compared for. A pair of numeric items can be compared to see which is greater, while a pair of uninterpreted items can be compared only for equality. In general, a concept of "greater than" is builtin for a datatype only if it is a very widely applied concept. The comparison operators are used in the construction of inclusion conditions when defining subsets of aggregate data.

The result of a comparison operation is a boolean item: one whose value is either TRUE or FALSE. Logical primitives are provided and generalized boolean functions can be constructed from them. With logical and comparison operators, complex conditions for inclusion of objects in sets can be specified.

Arithmetic operators will be available for the manipulation of numeric data. Here, we are not interested in generalized computation, but in applications of arithmetic in data selection, space allocation, subscript calculation, iteration control, etc.

Conversion is an important part of generalized data translation, and we are interested in providing a substantial builtin conversion facility. In particular, we will want to provide an efficient system routine for each "standard" or widely-used conversion function. Of particular importance are conversions to and from character string data; in character string representation of, for example, numeric items, there are many possible formats corresponding to a single data type. Conversion between character sets and dealing with padding and truncation are viewed as conversion problems.

There are two principal classes of primitive operators defined on aggregates: those related to data reference (see previous section) and thos which add and delete components. Changing an existing component is accomplished through assignment, and is an operation on the component, not the aggregate.

Addition and deletion of components is defined only for aggregates which are not inherently static in composition. Thus one can add a component to a LIST, but to an ARRAY. To specify deletion it is necessary to specify which component is to be deleted, and from which aggregate (in the case that it is shared). Addition requires specification of new component, aggregate, and sometimes auxiliary information. For example, some aggregate types would permit addition of new components anywhere in the structure; in these a postion must be indicated, relative to any existing components.

Often it is desirable to operate on some of the members of a list, or to treat a group of members as a list in its own right. For example, it might be common to transmit to a remote program for analysis, the medical history of patients developing heart disease before the age of 30. These may be just a few of the members of a large list of patients.

In this case, the operation to be performed is transmission to the remote system; this operation is performed on a several members of the list of patients. The ones to be transmitted are thought of as a set; the set is specified as containing all the members of a given list satisfying two conditions: (1) age less than 30, and (2) has heart disease.

Sets can be defined explicitly, or implicitly simply with appropriate reference mechanisms. Definition of a set is distinct from identification of membership, which is distinct from access to membership. Definition involves specifying the candidates for set membership and specifying a rule by which members of the set can be distinguished from non-members; for example, an inclusion condition such as "under 30 with heart disease". Identification involves effective application of the rule to all candidates for membership. When the membership has been identified, it can be counted, but the data itself has not necessarily been accessed. When a member or member is accessed, its contents can be operated on.

Primitives to accomplish each of these operations on a set will be provided; however, it will ordinarily be optimal for the datacomputer to determine when each step should be performed. To enable users to operate at a level at which the datacomputer can optimize effectively, higher-level operators on sets will be provided. Some of these are logical operators, such as union and intersection. These input and output sets. Also available is an operator which complements a set (since the definition establishes all possible candidates, a set always has a well-defined complement).

These higher level operators can be applied to any defined set; the set members need not be identified or accessed. The system will perform such operations without actually accessing members if it can.

Some of the other operators on sets are counting membership, partitioning a set into a set of sets, uniting a set of sets into a set. A set can be used to reference another set, providing there is a well-defined way to identify members of the second set given the first set. For example, a set C may contain all the children doing poorly in school. A set F may be defined, where the members of F are the records about families having a child in set C.

Some other useful operations on sets are: adding all the members of a set to an aggregate, deleting all the members of a set (frequently such a massive change can be performed far more efficiently than the same set of changes individually requested), changing all the members of a set in a given way.

A set can be made into a list, by actually accessing each member and physically collecting them.

Some of the operations on lists are: concatenation of lists into larger
lists, division of a list into smaller lists, sorting a list, merging a pair
of ordered lists (preserving order).

This is not intended to be a full enumeration of high-level operations, but
to be suggestive. We are planning to build in high-level functions for
operations which are used very commonly, and can be implemented within the
system significantly better than they can be implemented by users in the
language. For most of the functions mentioned here, considerable knowledge
is accumulated on good implementations. In particular, the techniques used
for inverted file access provide many set operations to be performed without
actual access to the data.

3.14 Control

The control features of datalanguage are to the basic operations as data
aggregates are to the basic data items. Control features are used to create
complex requests out of the basic requests provided by datalanguage.

Conditional requests allow the user to alter the normal request flow by
specifying that certain requests are to be executed under certain conditions.
In general datalanguage will provide the ability to chose at most one of a
number of requests to be made based on some set of conditions or the value of
some item. In its simplest form the conditional allows for optional
execution of a given request.

Iterative requests cause a request (called the body) to be executed a fixed
or variable number of times or until a given condition is met. Datalanguage
will provide iterative requests that will allow for similar manipulations to
be performed on all members of some aggregate structure as well as the
standard type of iterative request based on counters. By providing a
capability of directly expessing manipulations on aggregates which require
processing all of the items subordinate to the aggregate, the datacomputer
can be more efficient in processing user requests. For example, a user
defined conversion process which operates on character strings, can be
implemented far more efficiently if the datacomputer is explicitly informed
that the process requires sequential processing of the characters.
Datalanguage will also provide for parallel iteration. For example, the user
will be able to specify operations which require sequencing through two or
more lists in parallel This would be done if the contents of one file wer
to be updated based on a file of correction information.

compound requests are collections of requests which act as one. They are
primarily provided to allow for the conditional performance of or iteration
on more than one statement. Compound requests also provide request reference
points which can be used to control the request processing flow. That is,
compound requests can be "named". The datalanguage user will be able to
specify control information which will conditionally cause a compound request
to be exited. By providing naming, the user may cause any number of
previously entered compound requests to be exited.

We do not intend to provide the traditional goto capability. By not
including a goto request, the chances for efficient operation (via
optimization) of the datacomputer are increased. We also hope, in this way,
to force the datalanguage user to specify his data manipulations in a clear
style.

Two forms of the compound request will be provided, ordered and unordered.
In the unordered case the user is informing the datacomputer that the
requests can be performed in any order. This should allow the datacomputer
to perform more efficiently and might even allow for parallel processing.

During a session with the datacomputer it is likely that a user will find a
need for temporary data. That is, data which is used to remember, for a
short term, information which is needed for the processing of requests. This
short term might be a session or a small part of a session. Datalanguage
will provide a temporary data facility. Temporary data will be easy to
create, use, and dispose of. This will be accomplished by allowing the
system to (optionally) make many decisions regarding the data. For example
the representation of a temporary integer item will often be of no concern to
the user. Some features which are provided for permanent data will be deemed
irrelevant with regard to temporary data.

Temporary data will be associated with a collection of requests in what will
be called a block. A block will be no different than a compound request with
the exception that data is defined with the requests which compose it and is
automatically created on entrance to the block and destrtoyed on exiting the
block.

3.15 Extensibility

The goals of datalanguage are to provide facilities of data structure at two
levels. At one level the user may take advantantage of high level data
capabilities which will do much of his data management work automatically and
which allows for the data computer to operate more effectively in some cases
since it has been given control of the data. At another level, however,
features are provided which allow the user to describe his application in
terms of primitive concepts. In this way the datacomputer user may compose a
large variety of data constructs and has great flexibility with respect to
the manipulations he can perform on his data. Also by interacting with the
datacomputer at the primitive level, the user can exercise a good deal of
control over tne methods employed by the datacomputer which may result in
more effective usage of resources for non-standard applications.
Datalanguage will provide features which allow the user to create an
environment whereby the datacomputer systems appears to provide features
especially tailored to his application.

The control features discussed above allow the user to extend the operations
available on data by appropriate composition of the operations. Datalanguage
will provide a method of defining a composite request to be a new request
(called a function). In this way a new operation on specific data can be
defined once and then used repeatedly. In order that the user may define
general operations, datalanguage will provide functions which can be

parameterized. That is, functions will not only be able to operate on specific data but may be defined to work on any data of a specific type. This capability will not be limited to basic data types (eg. integers) or even specific aggregate types (eg. array of integers) but will also include the ability to define functions which operate on classes of data. For example, functions can be defined which operate on lists independent of the type of the list members. Also provided, will be the abiltiy to expand and modify existing functions as well as creating new functions. This includes expanding the types of data for which a function is defined or modifying the behavior of a function for certain types of data.

As with operations, the data aggregates discussed above allow the user to extend the primitive data types by appropriate composition. For example, a two dimensional array of integers can be created by creating an array of arrays of integers. The situation for data types is analogous to that of operations. Datalanguage will provide the ability to define a composition of data to be a new data type. Also the capability of defining general data structures will be provided by essentially parameterizing the new data definition. This would allow the general concept of two dimensional array to be defined as an array of arrays. Once defined, one could create two dimensional arrays of integers, two dimensional arrays of booleans, etc. As with functions there is also a need to expand or modify existing data types. One might want to expand the attributes which apply to a given data type, in that he might want to add new attributes, or add new choices for the existing attributes.

The control features can be extended also. Special control features might be needed to process a data structure in a special way or to process a user defined data structure. For example, if a tree type data structure has been defined in terms of lists of lists, the user might like to define a control function which causes a specified operation to be performed on each item of a specified tree. As with data types and functions, there is a need to be able to modify and extend existing control features as well as the ability to create new ones.

Datalanguage will provide the ability to treat data descriptions and operations in much the same way that data is treated. One can describe and manipulate descriptions and manipulations in the same way that he can describe and manipulate data. It is impossible to talk about data types without consideration of operations and equally as impossible to talk about operations without an understanding of the data types they operate on. In order for the user to be able to effect the behavior of the datacomputer system, the design of datalanguage will include a definition of the operational cycle of the datacomputer. Precise definitions of all aspects of data (data attributes, data classes, relationship of aggregates to their subordinate items, etc.) in terms of their interaction with datalanguage operations will be made. In this way the datacomputer can offer tools which will give the datacomputer user the ability to be an active participant in the design of the datalanguage which he uses.

4. A Model for Datalanguage Semantics

For the purpose of defining and experimenting with language semantics and with language processing techniques, we are developing a model datacomputer.

The principal elements of the model are the following:
(1) A set of primitive functions
(2) An environment in which data objects can be created, manipulated and deleted, using the primitives
(3) A structure for the representation of collections of data values, their descriptions, their relationships, and their names.
(4) An interpreter which executes the primitives
(5) A compiler which inputs requests in a very simple language, performs binding and macro expansion operations, and generates calls to the internal semantic primitives.

If our modeling efforts are successful, the model will evolve until it accepts a language like the datalanguage whose properties we have described in sections 2 and 3 of this paper. Then the process of writing the final specification will simply require reconciliation of details not modeled with structure that has been modeled. One rather large "detail" which we may never handle within the model is syntax; in this case reconciliation will be more involved; however, we firmly believe that the semantic structure should determine the syntax rather than the opposite, so we will be in the proper position to handle the problem.

By constructing a model for each of the elements listed above, we are "implementing" the language as we design it, in a very loose sense. In effect, we work in a laboratory, rather than working strictly on paper. Since we aren't concerned with the performance or usability of the datacomputer we are building in the laboratory, we are able to build without becoming involved with some of the most time-consuming concerns of an implementor. However, because we are building and tinkering, rather than simply working on paper, we do get some of the advantages that normally come with the experience of implementing one's ideas.

The model datacomputer is a program, developed in ECL, using the EL1 language. Presently we are interested in the process of developing the program, not running it. Our primary requirement is to have, in advance of the existence of datalanguage, a well-defined and flexible notation in which to specify data structures, function definitions and examples. EL1 is convenient for this. Having a program which actually works and acts like a simple datacomputer is really a by-product of specifying semantics in a programming language. It is not necessary for the program to work, but it does provide some nice features. It enhances the "laboratory" effect, by doing such things as automatically compiling strings of primitives, displaying the state of the environment in complicated examples, automatically discovering inconsistencies (in the form of bugs½), and so on.

There are two major reasons that EL1 is a convenient notation for specifying datalanguage semantics. One is that the languages have a certain amount in common, in both concepts and in goals in data description. (In part, this is

because EL1 itself has been a good source of ideas in attacking the datalanguage
problem). Both languages emphasize operations on data, independent of underlying
representation.   A second reason that EL1 is a convenient way to specify
datalanguage, is that EL1 is extensible; in fact, many primitive functions could
be embedded directly into EL1 by using the extension facilities. At times, we
have chosen to embed less than we could, to expose problems of interest to us.

So far, the model has been useful primarily in exposing design issues and
relationships between design decisions. Also, because it includes so many of the
elements of the full system (compiler, interpreter, environment, etc.), it
encourages a fairly complete analysis of any proposal.

In presenting the model in this section, we have chosen to emphasize ideas and
examples, rather than formal definitions in EL1. This is because the ideas are
more permanent and relevant at this point (the formalisms are changing rather
frequently) and because we imagine people reading the formal definitions only to
get at the ideas. The formal definitions may be interesting in themselves when
the language is complete; at this point they are probably of interest only to us.

The section is organized into a large number of sub-sections. The first few are
concerned with the basic concepts of data objects, descriptions, and
relationships between objects. We then discuss primitive semantic functions and
present informal definitions and examples in sections 4.7 and 4.8. Section 4.9
is a brief discussion of compilation, interpretation and the execution cycle.
Section 4.10 provides a fairly elaborate example of how primitive functions can
be combined to do something of interest: a selective retrieval by content.   The
last two sections wrap up with discussions of high-level functions and some
conclusions.

4.1 Objects

An object has a name, a description, and a value. It can be related to other
objects.

The name is a symbol, which can be used to access the object from language
functions.

The description is a specification of properties of the object, many of which
relate to the meaning or the representation of the value.

The value is the information of ultimate interest in the object.

The relationships between objects are hierarchical.   Each object can be
related directly to at most four other objects, designated as its parent, its
child, its left sibling, and its right sibling.

This specific concept of relationship is all that has been built in to the
model to date. One of our primary objectives in the future is to experiment
with more general relationships among objects.

4.2 Descriptions

A _description_ has the components _name_, _type_ and _type-dependent parameters_. It can be related hierarchically to other descriptions, according to a scheme similar to the one described for objects in 4.1.

The _name_ has a role in referencing, as in the case of objects.

Type is an undefined, intuitive idea for which we expect to develop a precise meaning within datalanguage(see section 3.10 for some of the ideas about this). In terms of the present model, it simply means one of the following: LIST, STRUCT, STRING, BOOL, DESC, DIR, FUNC, OPD. Each of these refers to a sort of value corresponding to common ideas in programming (with the exception of OPD, which is explained in section 4.7), and on which certain operations are defined.

Examples of _type-dependent parameters_ are the two items needed to define a STRING: size option and size. A STRING is a sequence of characters; the size of the STRING is the number of characters in it. If a STRING has a fixed size, then size option is FIXED and size is the number of characters it always contains. If a STRING has a varying size, then size option is VARYING, and size is its maximum (clearly, it might also have a minimum in a more refined scheme).

When the description of an object has a type of STRING, it is commonly said that the object is a STRING.

4.3 Values

The value is the data itself.

An object of type BOOL can have only either the value TRUE or the value FALSE.

An object of type STRING has values such as 'ABC', 'JOHN', or 'BOSTON'.

Each value has a representation, in bits. Thus a BOOL is represented by a single bit, which will be a 'one' to represent TRUE and a 'zero' to represent FALSE.

4.4 Some examples

Here are some examples of structures involving objects, descriptions, and values. In these explanations and drawings, the objective is to convey some ideas about these primitive structures; considerable detail is omitted in the drawings in the interest of clarity.
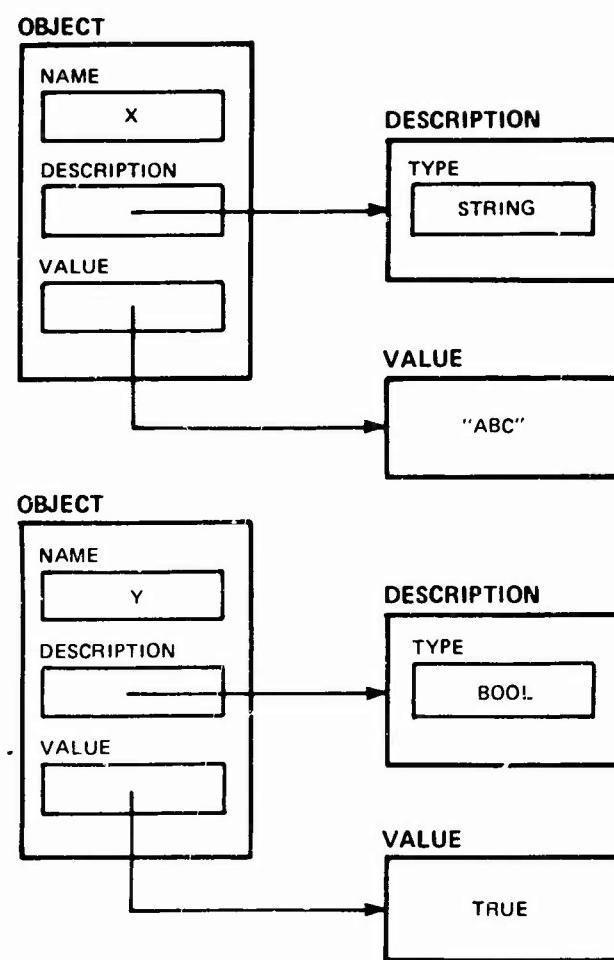
Figure 4-1
Two elementary objects

Figure 4-1 shows two objects. X is of type string and has value 'ABC'. Y is of type bool and has value TRUE.
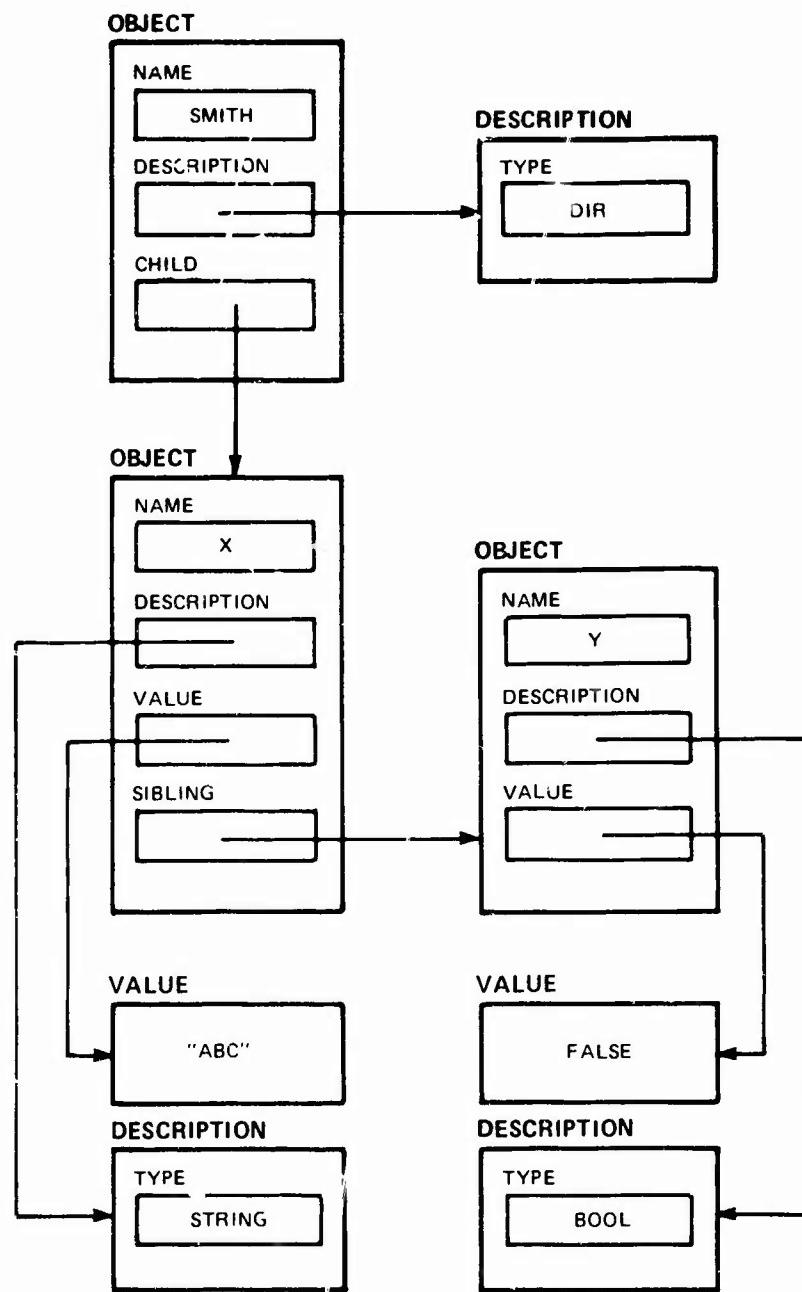
Figure 4-2
A directory with two members

Figure 4-2 illustrates an object of type dir (a directory) and related
objects. The directory has name SMITH. There are two objects entered in
this directory, named X and Y.

The idea of a dir is similar to the idea of a file directory in most systems.
A directory is a place where one can store named objects, freely adding and

deleting them. The entries in the directory are all objects whose parent is that directory. Figure 4-3 shows a more rigidly structured group of objects. Here we have R. ≙ struct, and A and B, a pair of strings.   Note that the boxes labelled 'object' in figure 4-3 bear precisely the same relationships to one another as those labelled 'object' in 4-2.   However,   there are two conditions which hold for 4-3 but do not hold for 4-2: (1) the value of R contains the values of A and B, and (2) the descriptions of R, A and B are all related.

Structs have the following properties: (1) name and description of each component in the struct is established when the struct is created, and (2) in a value of the struct, the order of occurrence of component values is fixed.
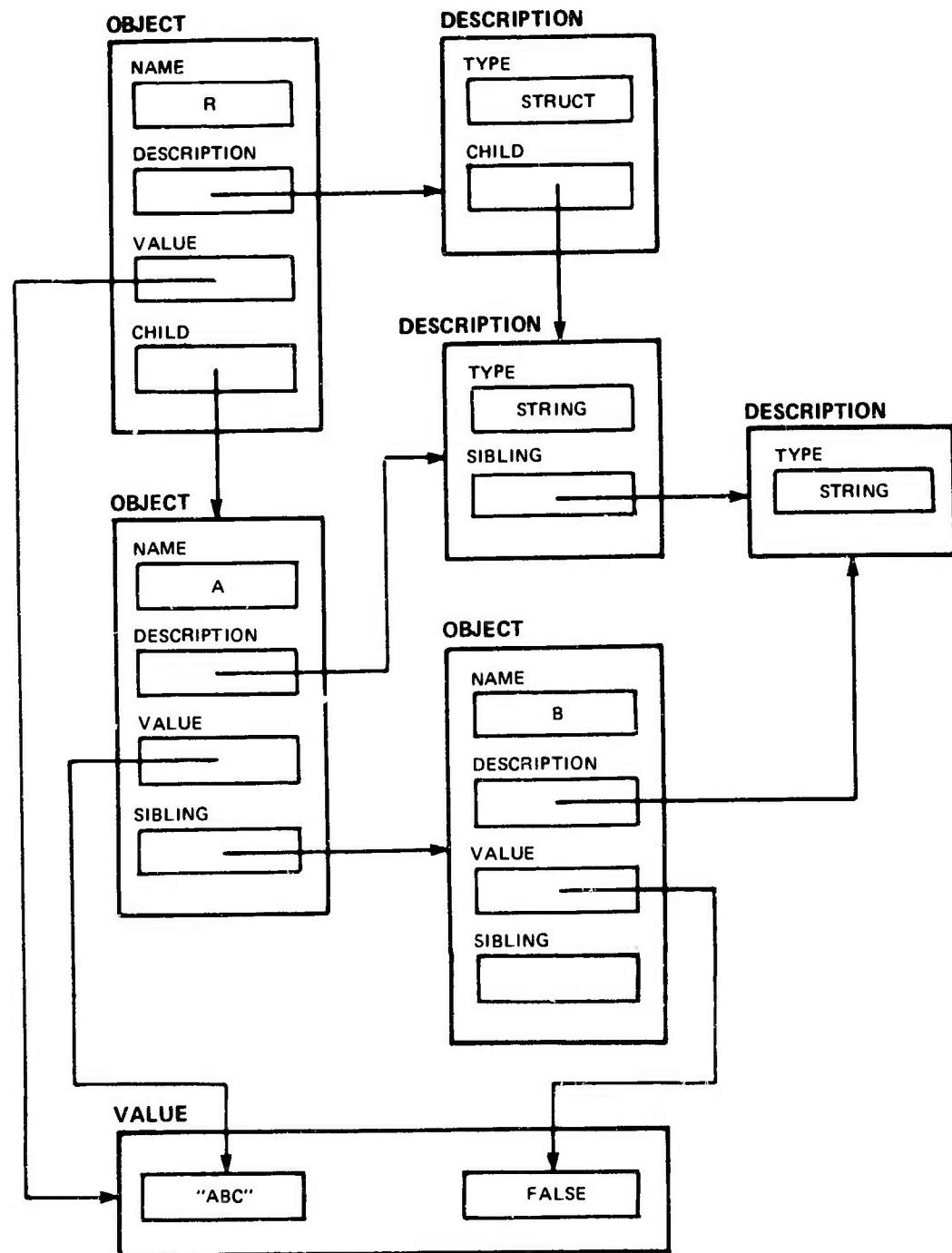
Figure 4-3
A STRUCT with two members
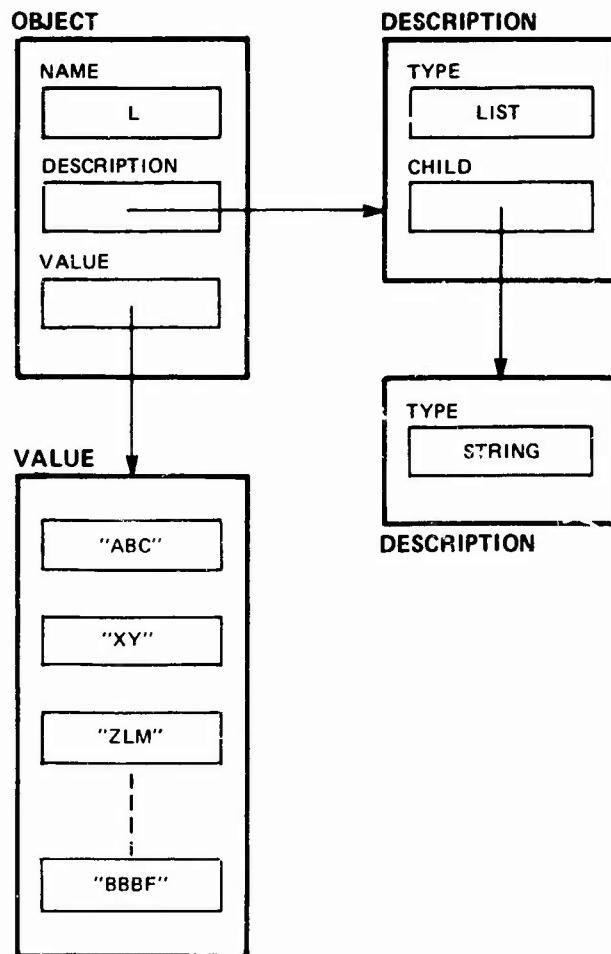
Figure 4-4
A LIST

Figure 4-4 shows a list named L. Here a similar structure of objects is implied, but because of the regularity of the structure, not all the boxes labelled 'object' are actually present.

L has a variable number of components, all satisfying the description subordinate to L's description.

We could imagine an 'object' box for each string in L. Each of these boxes would point to its respective string and to the common description of these strings. Instead, we think in terms of creating such boxes as we need them.

4.5 Definitions of types

Following are some more precise definitions of types, in terms of the present model. These serve the purpose of establishing more firmly the semantics of our structure of objects, descriptions and values; however, they should not be thought of as providing a definition for the completed language specification.

An object of type STRING has a value which is a sequence of characters (figure 4-1).

An object of type BOOL has a value which is a truth value (TRUE or FALSE--figure 4-1).

An object of type DIR has subordinate objects, each having its own description and value. Subordinate objects can be added and deleted at will (figure 4-2).

An object of type STRUCT has subordinate objects, each of which has a description which is subordinate to the STRUCT's description, and a value contained in the STRUCT's value. The number, order and description of components is fixed when the STRUCT is created (figure 4-3).

An object of type LIST may be thought of as having imaginary subordinate objects, whose existence is simulated by the use of appropriate techniques in processing the LIST. Each of these has the same description, which is subordinate to the description of the LIST. Each has a distinct value, contained in the value of the LIST. In fact, only the LIST object, the LIST and component descriptions, and the values exist. (figure 4-4)

An object of type DESC has a description as its value. This value is the same sort of entity which serves as the description of other objects.

An object of type FUNC has a function call as its value. We will be able to say more about this after functions have been discussed.

An object of type OPD has an operation descriptor as its value. (see 4.7 for details).

4.6 Object environment

There are three categories of objects in the model datacomputer. These are p/objects, t/objects, and i/objects.

P/objects are permanent objects created explicitly with language functions. They correspond to the idea of stored data in the real datacomputer.

OBJECT

NAME

STAR

DESCRIPTION

DESCRIPTION

TYPE

DIR

CHILD
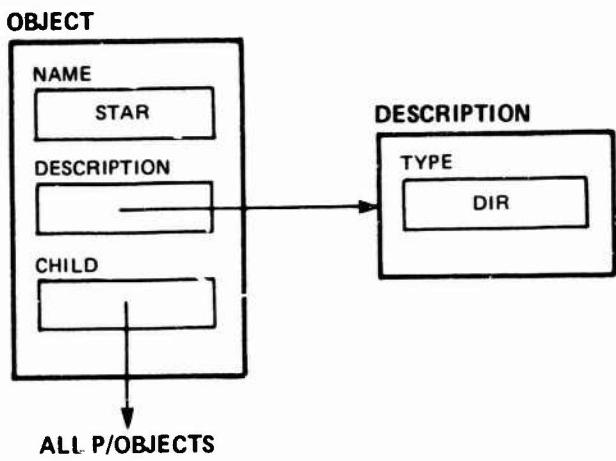
ALL P/OBJECTS

Figure 4-5
STAR and p/objects

T/objects are temporary objects, also created explicitly with language functions. However, these correspond to user-defined temporaries, both local to requests and "top-level" (i.e. not local to any request, but existing until deletion or logout.)
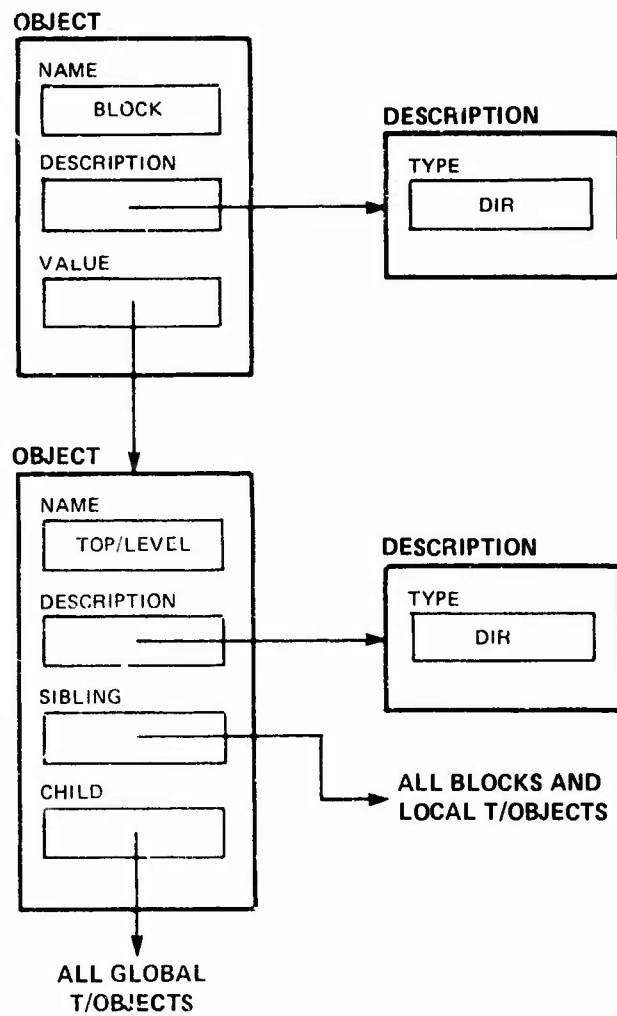
Figure 4-6
BLOCK, TOP/LEVEL and t/objects

I/objects are internal, system-defined objects whose creation and deletion is implicit in the execution of some language function.

There are three special objects. These are special only in that they are created as part of initializing the environment, rather than as the result of executing a language fuction. These are named STAR, BLOCK and TOP/LEVEL. All three are of type DIR.

An object is a p/object if it is subordinate to star; it is a t/object if it is subordinate to BLOCK. TOP/LEVEL is subordinate to BLOCK. (see figures 4-5 and 4-6).

I/objects are hung directly off of function calls (objects of type FUNC), and
are always local to the execution of such function calls. They correspond to
the notions of (1) literal, and (2) compiler- or interpreter-generated
temporary.
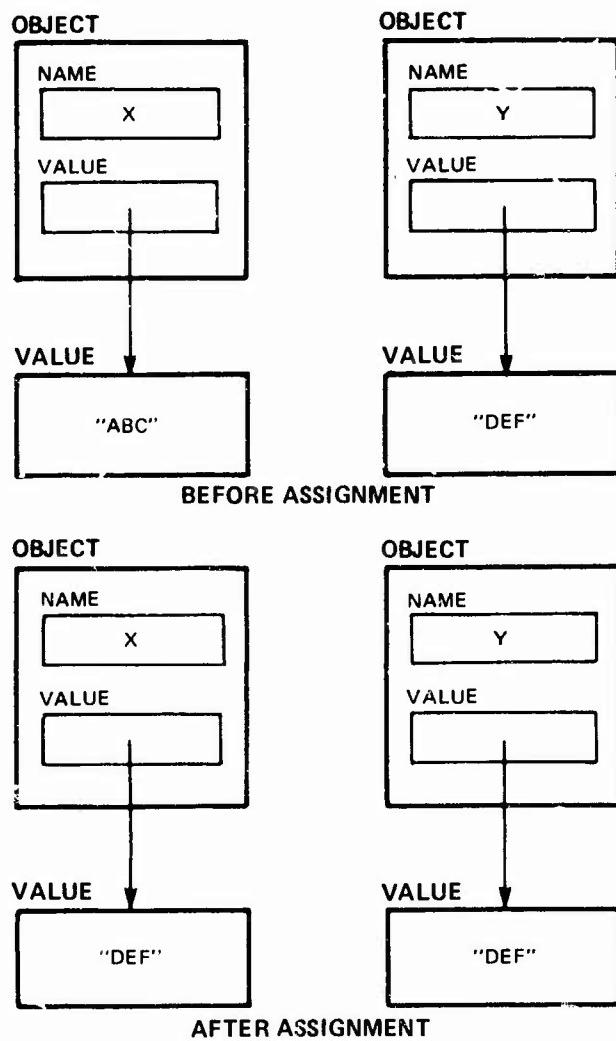
Figure 4-7
Effect of assignment

4.7 Primitive Language Functions

Here we discuss the primitive language functions presently implemented in the model and likely to be of most interest. In this section, the emphasis is on relating functions to one another. Section 4.8 contains more detail and examples.

Assign operates on a pair of objects, called the target and the source. The value of the source is copied into the value of the target. Figure 4-7 shows a pair of objects, X and Y, before and after execution of an assignment having X as target and Y as source. Presently, assignment is defined only for objects of type BOOL and objects of type STRING. The objects involved must have identical descriptions.

A class of primitive functions for manipulating LISTs is defined. These are called listops. All listops input a special object called an operation descriptor or OPD.

To accomplish a complete operation on a LIST, a sequence of listops must be executed. There are semantic restrictions on the composition of such sequences, and it is best to think of the entire sequence as one large operation. The state of such an operation is maintained in the OPD.

Refer back to figure 4-4. There is one box labelled "object" in this picture; this box represents the list as a whole. To operate on any given member, we need an object box to represent that member. Figure 4-8 shows the structure with an additional object box; the new box represents one member at any given moment. Its value is one of the components of the LIST value; its description is subordinate to the LIST description. In 4-8, the name of this object is M.

In 4-8 we have enough structure to provide a description and value for M, and this is sufficient to permit the execution of operations on M as an item. However, there is no direct link between object M and object L. The structure is completed by the addition of an OPD, shown in figure 4-9.
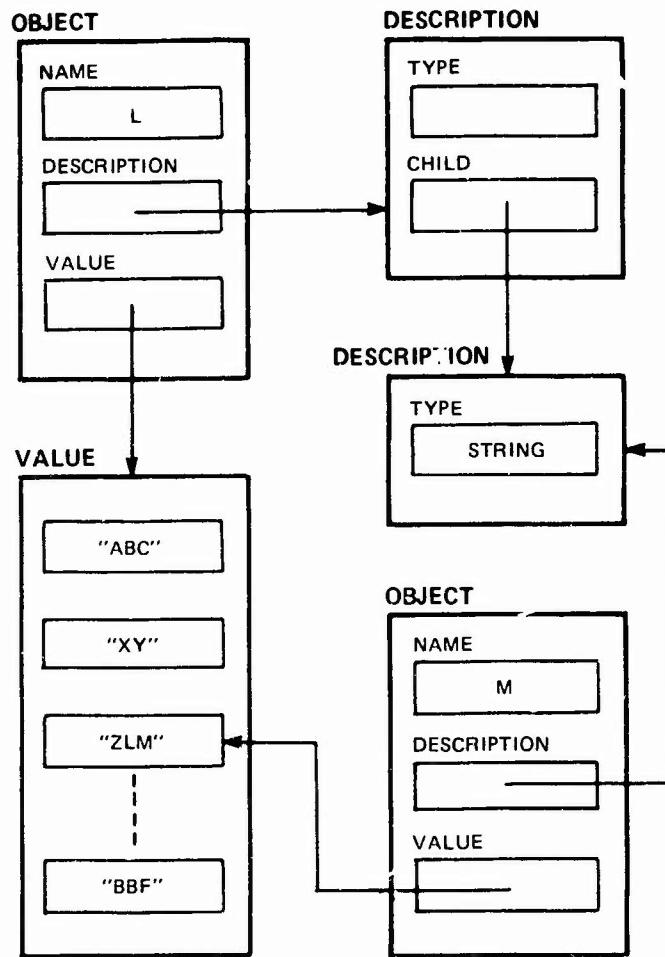
Figure 4-8
LIST and member objects
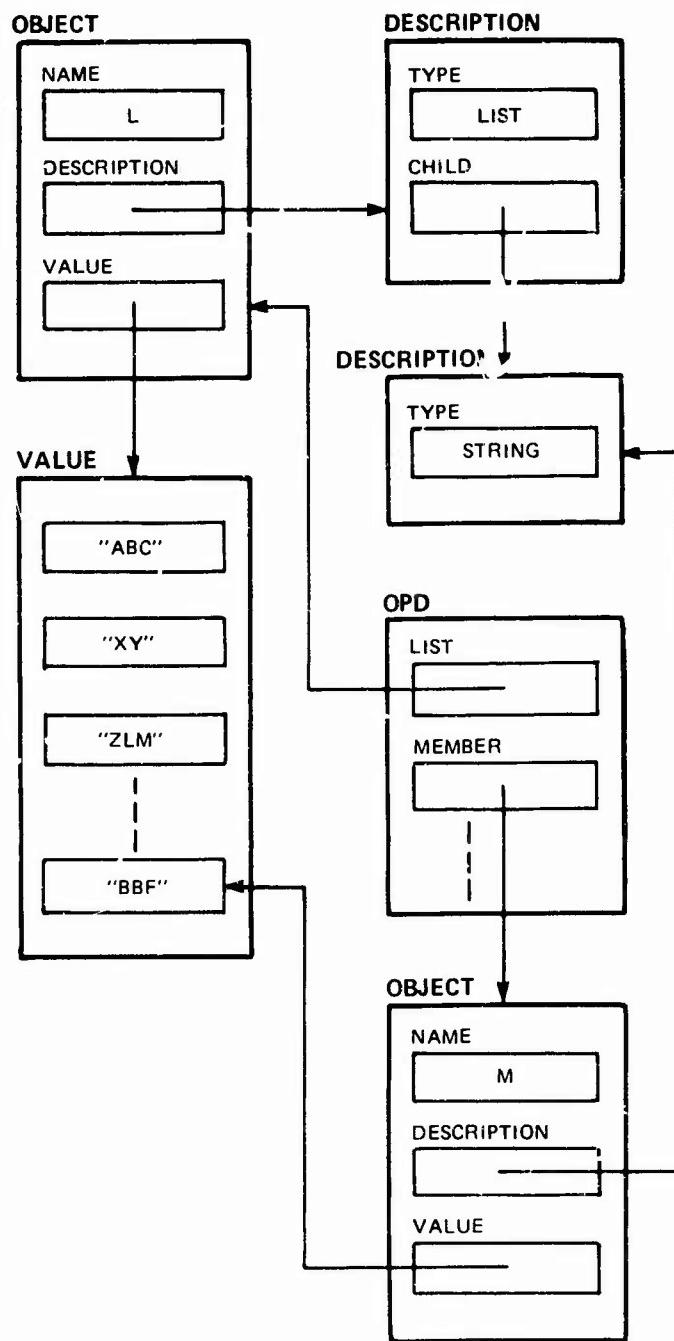
Figure 4-9
OPD, LIST and member

The OPD establishes the object relationship, and contains information about
the sequence of primitive listops in progress. When sufficient information
is maintained in the OPD, we have in 4-9 a structure which is adequate for
the maintenance of the integrity of the LIST and of the global list
operation. In addition to LIST and member pointers, the OPD contains
information indicating: (1) which suboperations are enabled for the sequence,
(2) the current suboperation, (3) the instance number of the current LIST
member, (4) an end-of-list indicator. The suboperations are add/member,
delete/member, change/member and get/member. All apply to the current
member. Only suboperations which have been enabled at the beginning of a
sequence may be executed during that sequence; eventually, the advance
knowledge of intentions that is implied by this will provide important
information for concurrency control and optimization.

Presently, an OPD relates a single member object to a single LIST object.
This imposes an important restriction on the class of operation sequences
which can be expressed. Any LIST transformation requiring simultaneous
access to more than one member must be represented as more than one sequence.
(And we do not yet solve the problems implied in concurrent execution of such
sequences, even when both are controlled by one process.)

Any transformation of a LIST can still be achieved by storing intermediate
results in temporary objects; however, it is certainly more desirable to
incorporate the idea of multiple current members into the semantics of the
language, than it is to use such temporaries. An important future extension
of the listops will deal with this problem.

There are six listops: listop/begin, listop/end, which/member, end/of/list,
open/member and close/member.

Listop/begin and listop/end perform the obvious functions of beginning and
terminating a sequence of listops. Listop/begin inputs LIST and member
objects, an OPD, and a specification of suboperations to enable. It
initializes the OPD, including establishment of the links to LIST and MEMBER
objects. After the OPD-LIST-member relationship has been established, it is
only necessary to supply the OPD and auxiliary parameters as input to a
listop in the sequence. From the OPD everything else can be derived.

Listop/end clears the OPD and frees any resources acquired by listop/begin.

Which/member establishes the current member for any suboperations. This is
either the first LIST member, the last LIST member, or the next LIST member.
This listop merely identifies which member is to be operated on; it does not
make the contents of the member accessible.

Open/member and close/member bracket a suboperation. The suboperation is
indicated as an argument to open/member. Open/member always establishes a
pointer from the member object to the member value; close/member always
clears this pointer. In addition, each of these listops make take some
action, depending on the suboperation.

The details of the action would be dependent on the representation of the
LIST in storage, the size of a LIST member, and choices made in

implementation.

Between execution of the open/member and the close/member, the data is accessible. It can always be read; in the case of the add/member and change/member suboperations, it can also be written into.

End/of/list tests a flag in the OPD and returns an object of type BOOL. The value of the object is the same as the value of the flag; it is TRUE if a get/member, change/member or delete/member would be unsuccessful due to a which/member having moved "beyond the end". This listop is provided so that it is possible to write procedures which terminate conditionally when all members have been processed.
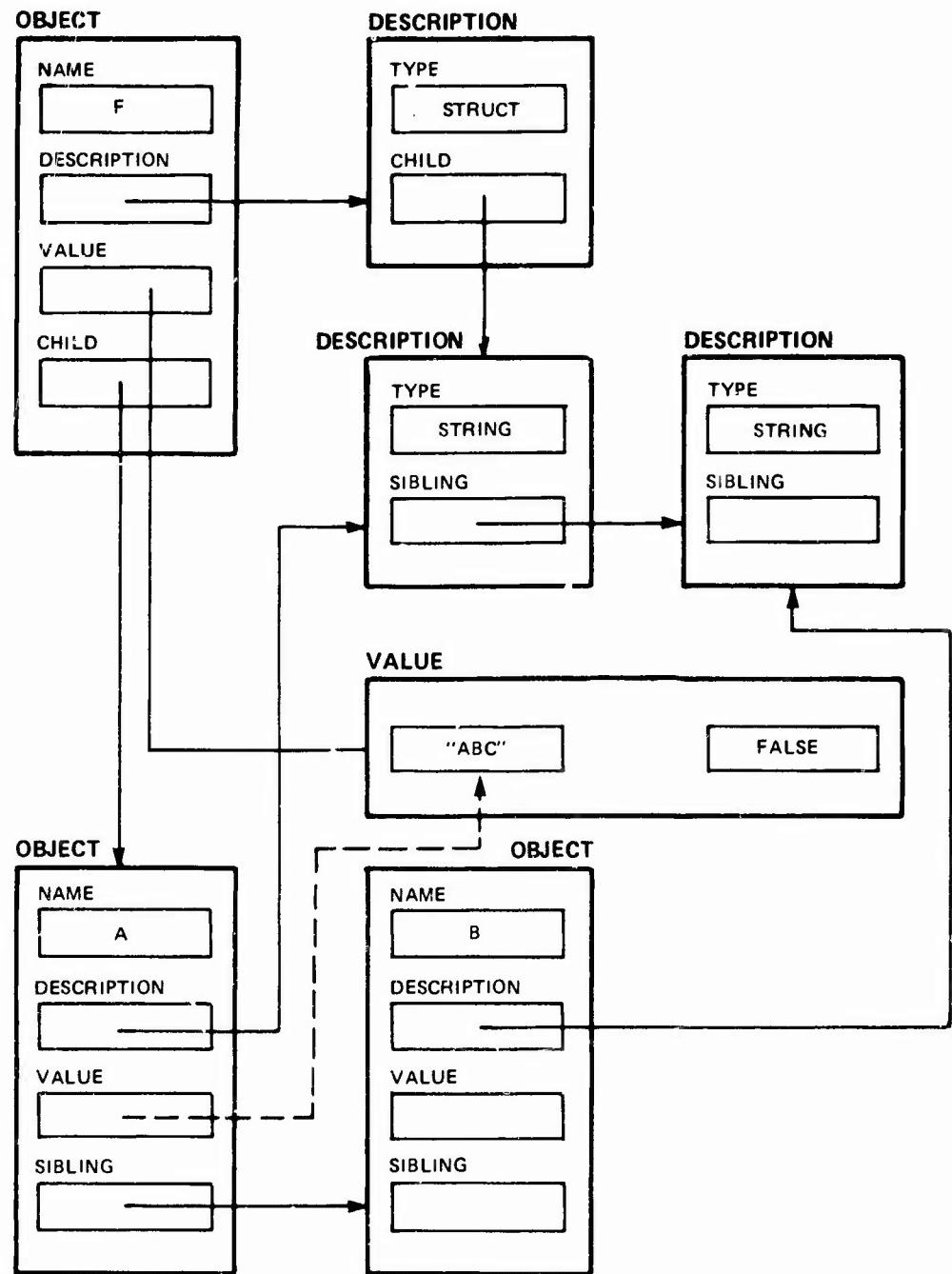
Figure 4-10
Effect of GET/STRUCT/MEMBER

Get/struct/member provides the ability to handle STRUCTs.   Given a STRUCT
object which points to the STRUCT value, it will establish a pointer from a
given member object to the member value.   (The pointer it establishes is
represented by a dashed line in figure 4-10).

The primitives discussed so far (assign, listops, and get/struct/member)
provide a basic facility for operating on structures of LISTs, STRUCTs and
elementary items. Using only them, it is possible to transfer the contents
of one hierarchical structure to another, to append structures, to delete
portions of structures, and so on. To perform more interesting operations
facilities for control and selection are needed.

A rudimentary control facility is provided through the primitives if/then,
if/then/else, till and while. All of these evaluate one primitive function
call, which must return a BOOL. Based on the value of this BOOL some action
is taken.

Let A and B be function calls. If/then(A,B) will execute B if A returns
TRUE.   If/then/else(A,B,C) will execute B if A returns TRUE; it will execute
C if A returns false. The while and till operators iterate, executing first
A then B.   While terminates the loop when A returns FALSE; till terminates
the loop when A returns TRUE. If this happens the first time, B is never
executed.

So far, we have mentioned one function which returns a BOOL: the listop,
end/of/list. Two other classes of functions which have this property are the
booleans and the comparisons. There are 3 primitive booleans (and, or, not)
and six primitive comparisons (equal, less/than, greater/than, not/equal,
less/than/or/equal, greater/than/or/equal -- only equal is implemented at
time of publication).

The booleans input and output BOOLs; the comparisons input pairs of
elementary objects having the same description and output BOOLs. Expressions
composed of booleans and comparisons on item contents are one of the
principal tools used in selectively referencing data in data management
systems.

With the booleans, the comparisons, and the primitives identified earlier, we
can perform selective "retrievals". That is, we can transfer to LIST B all
items in LIST A having a value of 'ABC'.   In fact, we now have a
(semantically) general ability to perform content-based retrievals and
updates on arbitrary hierarchical structures. We can even program something
as complex as the processing of a list of transactions against a master list,
which is one of the typical applications in business data processing.

Of course, we would not expect users of datalanguage to express requests at
the level of listops. Further, the listops defined here are not a very
efficient way of performing some of the tasks we have mentioned. To get good
solutions, we need both higher-level operators and other primitives which use
other techniques in processing.

In addition to those already discussed, the model contains functions for: (1)
referencing an object by qualified name, (2) generating a constant, (3)

generating data descriptions, (4) writing compound functions and blocks with local variables, (5) creating objects.

The facilities for generating constants and data descriptions (which are a special case of constants) are marginal, and have no features of special interest. Obviously, data description will be an important concern in the modeling effort later on.

Object referencing functions permit reference to t/objects and p/objects (these terms are defined in 4.6). A p/object is referenced by giving the pathname from STAR to it. A t/object is referenced by giving the pathname from the block directory in which it is defined to it.

Compound/function permits a sequence of function calls to be treated syntactically as a single call. Thus, for example, in if/then(A,B), B is frequently a call to compound/function, which in turn calls a sequence of other functions.

Create takes two inputs: a superior object and a description. The superior must be a directory. The new object is created as the leftmost child of the directory, its name is determined by the description.

4.8 Details of primitive language functions

This section provides specifications for the primitives discussed in the previous section. We are still omitting details when we judge them to be of no general interest; the objective is to provide enough information for the reader to examine examples.

Most of the primitives occur at two levels in the model. The internal primitives are called i/functions and the external, or language primitives are called l/functions. The relationship between the two types are explained in 4.9. In this section we discuss l/functions.

L/functions input and output <u>forms</u>, which are tree structures whose terminal nodes are atoms. The atoms are such things as function names, object names, literal string constants, truth values and delimiters. Calls to l/functions are also expressed as forms.

Any form can be evaluated, yielding some object. A form which is an l/function call yields the value returned by the l/function: another form. In general, the form returned by an l/function call will, when evaluated, yield a datalanguage object (that is, the sort of object we have been representing by an "object box" in the drawings).

4.8.1 Name recognition functions

These return a form which evaluates to an object.

L/TOBJ

Input must name a temporary object subordinate either to TOP/LEVEL or a block directory.

L/POBJ

Input must name a permanent object (i.e., an object subordinate to STAR).

Typical calls are L/POBJ(X.Y.Z) and L/TOBJ(A).

4.8.2 Constant generators

Each of these inputs an atomic symbol yielding a value for a constant to be created. Each returns a form which will evaluate to an object having the specified value and an appropriate description.

LC/STRING - a typical call is LC/STRING('ABC')

LC/BOOL - a typical call is LC/BOOL(TRUE)

4.8.3 Elementary item functions

These input and output forms evaluating to elementary objects (objects which can have no subordinate object--in effect, objects whose value is regarded as atomic). Eventually all the comparison operators will be implemented.

L/ASSIGN

Inputs must evaluate either to STRINGs or BOOLs. Outputs a form which transfers the value of the second to the first. Typical call:
  L/ASSIGN(L/TOBJ(A),LC/STRING('XYZ'))
The output form, when evaluated, will copy 'XYZ' into A's value.

L/EQUAL

Inputs a pair of forms evaluating to objects, which must have identical descriptions and be BOOLs or STRINGs. Returns a form evaluating to an object of type BOOL. Value of this object is TRUE if inputs have identical descriptions and values; otherwise it is false. Typical call:
  L/EQUAL(L/TOBJ(X),LC/STRING('DEF'))

L/AND, L/OR, L/NOT

The standard boolean operators. Inputs are forms evaluating to BOOLs; output
is a form evaluating to a BOOL. L/AND and L/OR take two inputs; L/NOT takes
one. Typical call:
  L/AND(L/EQUAL(L/TOBJ(X),LC/STRING('DEF')),
                  L/EQUAL(T/TOBJ(Y),LC/STRING('GHI')))
The form returned will, when evaluated, return TRUE if both X has value 'DEF'
and Y has value 'GHI'.

4.8.4 Data description functions

These all return a form evaluating to a description (i.e. that which is
represented in our drawings by a box labelled "description").

LD/STRING

Inputs 3 parameters specifying the name, size option and size for the string.
Typical call:
  LD/STRING(X,FIXED,3)
This call returns a form evaluating to a description for a fixed-length
3-character string named X.

LD/LIST

Inputs two forms. The first is the name of the LIST and the second evaluates
to a description of the LIST member. Typical call:
  LD/LIST(L,LD/STRING(M,FIXED,3))
Creates the structure shown in figure 4-11, and returns a form evaluating to
the description represented by the upper box.
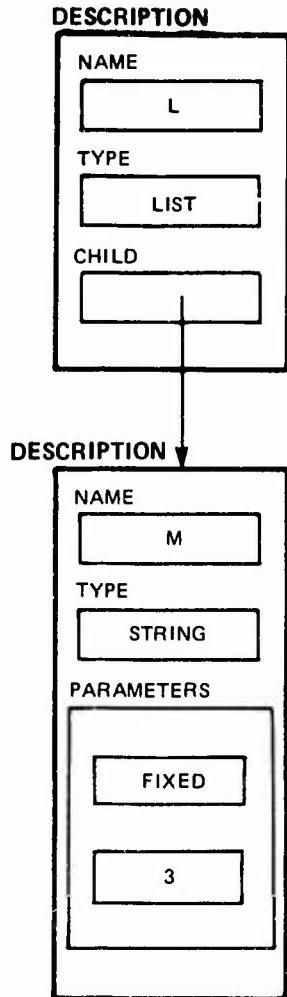
Figure 4-11
LIST and member descriptions

LD/STRUCT

Inputs a form to use as the name for the STRUCT and one or more forms
evaluating to descriptions; these are taken as the descriptions of the
members. Typical call:
```
LD/STRUCT(R,
        LD/STRING(A,FIXED,3),
        LD/BOOL(B) )
```
produces the structure shown in 4-12; returns a form evaluating to the top
box.
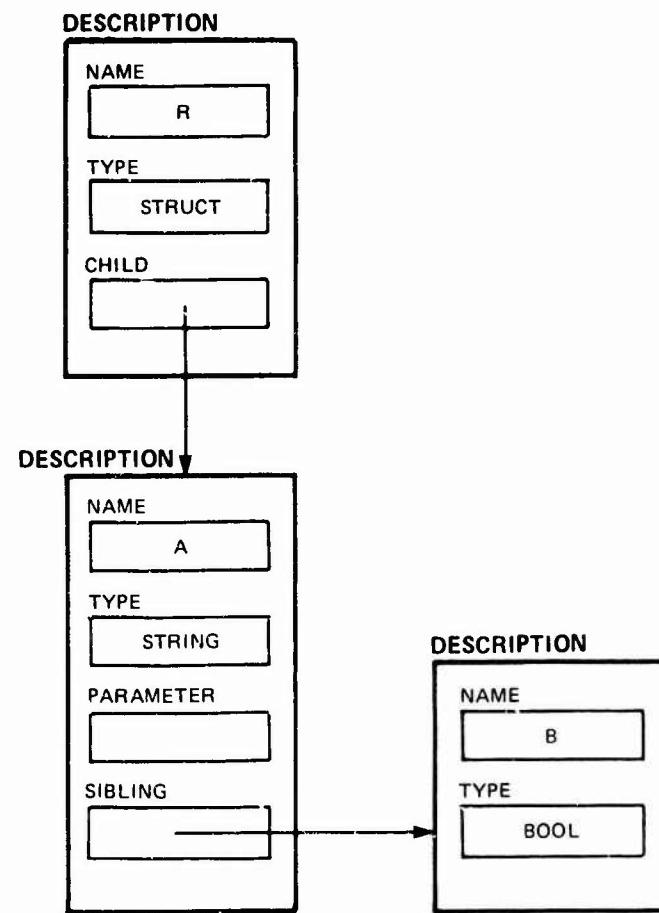
Figure 4-12
STRUCT and member descriptions

LD/BOOL, LD/DIR, LD/OPD, LD/FUNC, LD/DESC

Each inputs a name and produces a single description; each returns a form evaluating to the description produced. Typical call:
LD/BOOL(X)

4.8.5 Data creation

L/CREATE

Inputs two forms and evaluates them. First must yield an object of type DIR; second must yield a description for the object to be created. Creates the object and returns a form, which, when evaluated, will generate a value for

the new object. A simple example:
L/CREATE(L/TOBJ(X),LD/BOOL(Y))
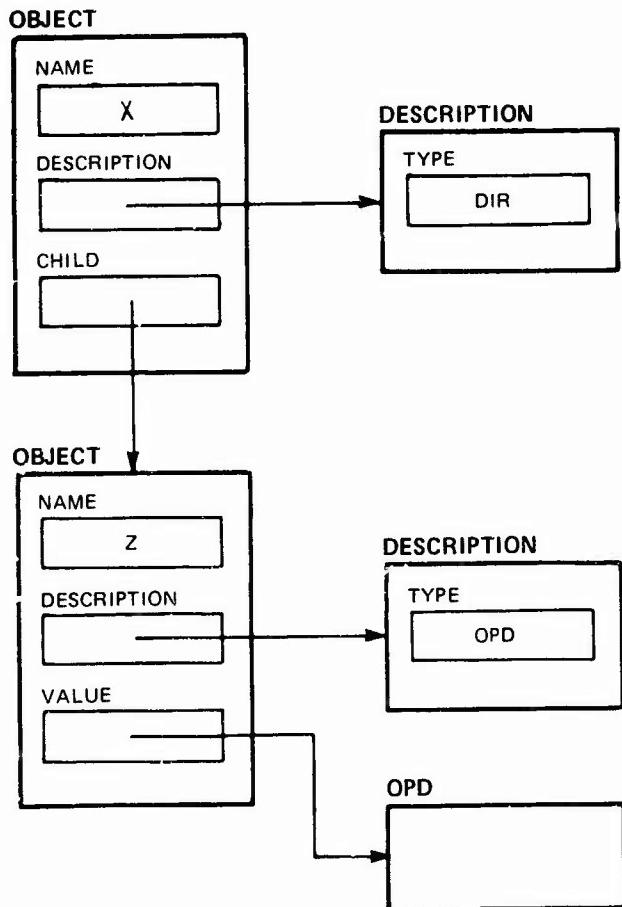
Figure 4-13
X and Z before creation of Y

Figure 4-13 shows the directory X before execution of the above call. It
contains only an OPD. After execution, the directory appears as in 4-14.
Creation of a value for Y occurs when the form returned by L/CREATE is
evaluated (covered in section 4.9).
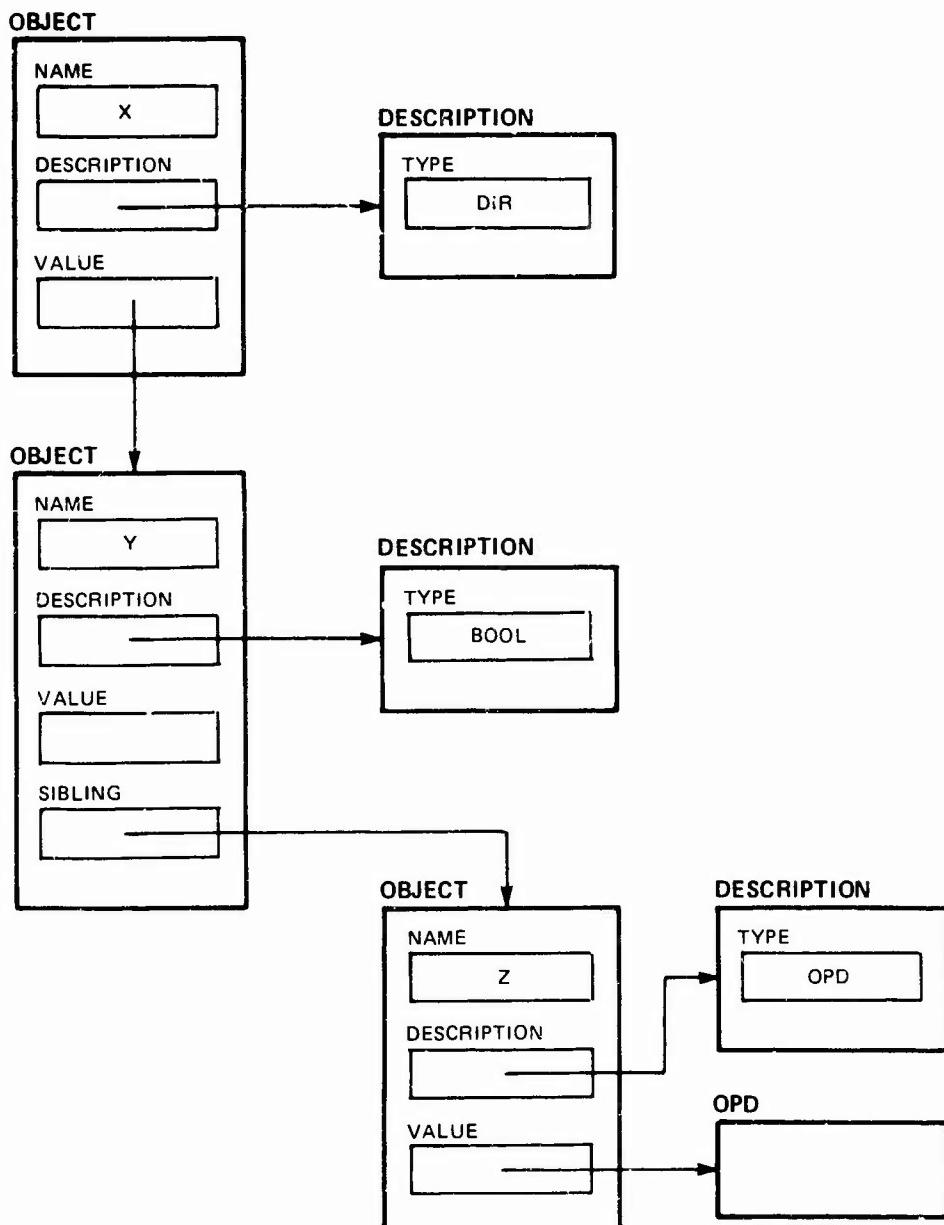
Figure 4-14
X, Y and Z after L/CREATE

4.8.6 Control

L/IF/THEN, L/IF/THEN/ELSE

Used to request conditional evaluation of a form. Typical call:
L/IF/THEN(L/EQUAL(L/TOBJ(A),LC/STRING('ABC'),
        L/ASSIGN(L/TOBJ(B),LC/STRING('DE')))
The form returned will do the following, when evaluated. if A has value 'ABC', then store 'DE' in the value of B.

L/WHILE, L/TILL

These iterate conditionally, as explained in the previous section. Examples appear later.

L/CF

Compound function: it inputs one or more forms and returns a form which, when evaluated, will evaluate each input in sequence. Typical call:
L/CF(L/ASSIGN(L/TOBJ(R.A),LC/STRING('XX')),
        L/ASSIGN(L/TOBJ(R.B),LC/STRING('YY')))
When the output of L/CF is evaluated, it will assign new values to R.A and R.B.

4.8.7 Listops

These primitives are executed in sequences in order to perform operations on LISTs. With the exception of L/END/OF/LIST these functions output forms which are evaluated for effect only; that is, the output forms do not themselves return values.

L/LISTOP/BEGIN

Inputs forms evaluating to: (1) a LIST, (2) an object to represent the current LIST member, (3) an OPD. Also, inputs a list of atomic forms whose values are taken as suboperations to enable. Typical call:
L/LISTOP/BEGIN(L/POBJ(F),L/TOBJ(R),
                L/TOBJ(OPF),ADD,DELETE)
This returns a form that will initialize a sequence of listops to be performed on F. Caller has previously created R and OPF. He intends to ADD and DELETE list members.

All subsequent calls in this sequence of listops need specify only the OPD and auxiliary parameters.

L/LISTOP/END

Inputs a form evaluating to an OPD. Outputs a form which, when evaluated, clears OPD and breaks relationships between OP: LIST and member objects.

L/WHICH/MEMBER

Inputs two forms. First evaluates to an OPD; second is one of FIRST, LAST,
NEXT. The form output, when evaluated, will establish a new current member
for the next suboperation. Note: this does not make the value of the member
accessible, it simply identifies it by setting the instance number in the
OPD. A typical call:
  L/WHICH/MEMBER(L/TOBJ(OPF),NEXT)
When a which/member causes advance beyond the end of the list, a flag is set
in the OPD.

L/END/OF/LIST

Inputs a form evaluating to an OPD. Outputs a form which, when evaluated,
returns a BOOL. This has value TRUE if the end of list flag in the OPD is
on.

L/OPEN/MEMBER

Inputs a form evaluating to an OPD and a form which must be one of ADD,
DELETE, GET, CHANGE. Outputs a form which, when evaluated, will initiate the
requested suboperation on the current LIST member. The suboperation always
establishes the pointer from the member object to the current member value
instance. In addition, in the case of ADD this value must be created.
Typical call:
  L/OPEN/MEMBER(L/TOBJ(OPF),ADD)

L/CLOSE/MEMBER

Inputs a form evaluating to an OPD. Outputs a form which, when evaluated,
will complete the suboperation in progress. A typical call:
  L/CLOSE/MEMBER(L/TOBJ(OPF))
Always clears the pointer from member object to member value. In addition,
in the case of DELETE, removes the member value from the LIST. In the case
of ADD enters the member value in the LIST. Makes the member added the
current member, so that a sequence of ADDs executed without intervening
which/members will add the new members in sequence.

An elaborate example, involving listops and several other primitives, appears
in section 4.10.

4.9 Execution cycle

The model datacomputer has a two-part execution cycle: it first compiles
requests, then interprets them. A "request" is an l/function call;
"compilation" is the aggregate result of executing all the l/function calls
involved in the request (typically this is many calls, as there are usually
several levels of nested calls, with the results of the inner calls being
delivered as arguments to the next level of calls). Usually, the process of
executing an l/function involves a simple macro expansion, preceded by some
binding, checking and (eventually) optimization.

The compiled form consists wholly of atomic symbols and i/function calls. The i/functions are internal primitives which input and output datalanguage objects (the entities represented by the boxes labelled "object" in the drawings).

Each of the l/functions discussed compiles into a single i/function; thus the macro expansion aspect of compilation is presently trivial. However, this will not be true in general; it is only that these are primitive l/functions that makes it true now.

The decision to use a compile-and-interpret cycle calls for some explanation. The way to understand this, is to think in terms of the functions that would be performed in a strictly interpretive system. There would still be a requirement to perform global checks on the validity of the request in advance of the execution of any part of it. This is because partial execution of an incorrect request can leave a database in an inconsistent state; if this is a large or complex database, the cost of recovery will be considerable. Thus it pays to do as much checking as is possible; when the system is fully developed, this will include a certain element of simple prediction of execution flow; in any case, much more than syntactic checking is implied.

Since any such global checks will be performed in advance of actual execution, they are effectively not part of the execution itself, for any given form. By performing them as part of a separate compilation process, we only formalize a modularity which already effectively exists.

There will still be cases, however, in which checking, binding and optimization functions must be executed during interpretation, if at all. This will occur when the information needed is not available until some of the data has been accessed. When practical, we will provide for such occurrences by designing most functions so that they can be executed as part of either "half" of the cycle.

As the model develops, we expect to get a better feel for this problem; it is certainly reasonable to end up with a structure in which there are many cycles of compilation and interpretation, perhaps forming a structure in which nesting of cycles within cycles occurs. 4.10 Examples of operations on LISTs Here we develop an example of an operation on a LIST using primitive l/functions. We first show the function calls required to create a LIST named F and give it a few member values. We then selectively copy certain members to a second LIST G.

To create F:

```
L/CREATE("STAR",LD/LIST(F,
                LD/STRUCT(R,
                        LD/STRING(A,FIXED,2),
                        LD/STRING(B,FIXED,2))))
```

This creates F as a member of the permanent directory STAR(see section 4.6 for details about STAR). The symbol STAR has a special status in the "language", in that it is one of the few atomic symbols to evaluate directly

to an object (recall that most permanent objects are referenced through a call to L/POBJ; reserving the symbol STAR is equivalent to reserving STAR as a name and writing L/POBJ(STAR). The solution we choose here is easier to write.) Execution of this call builds the structure shown in 4-15 (except for STAR, which existed in advance of the call). The value initially created for F is an empty LIST--a LIST of zero members.
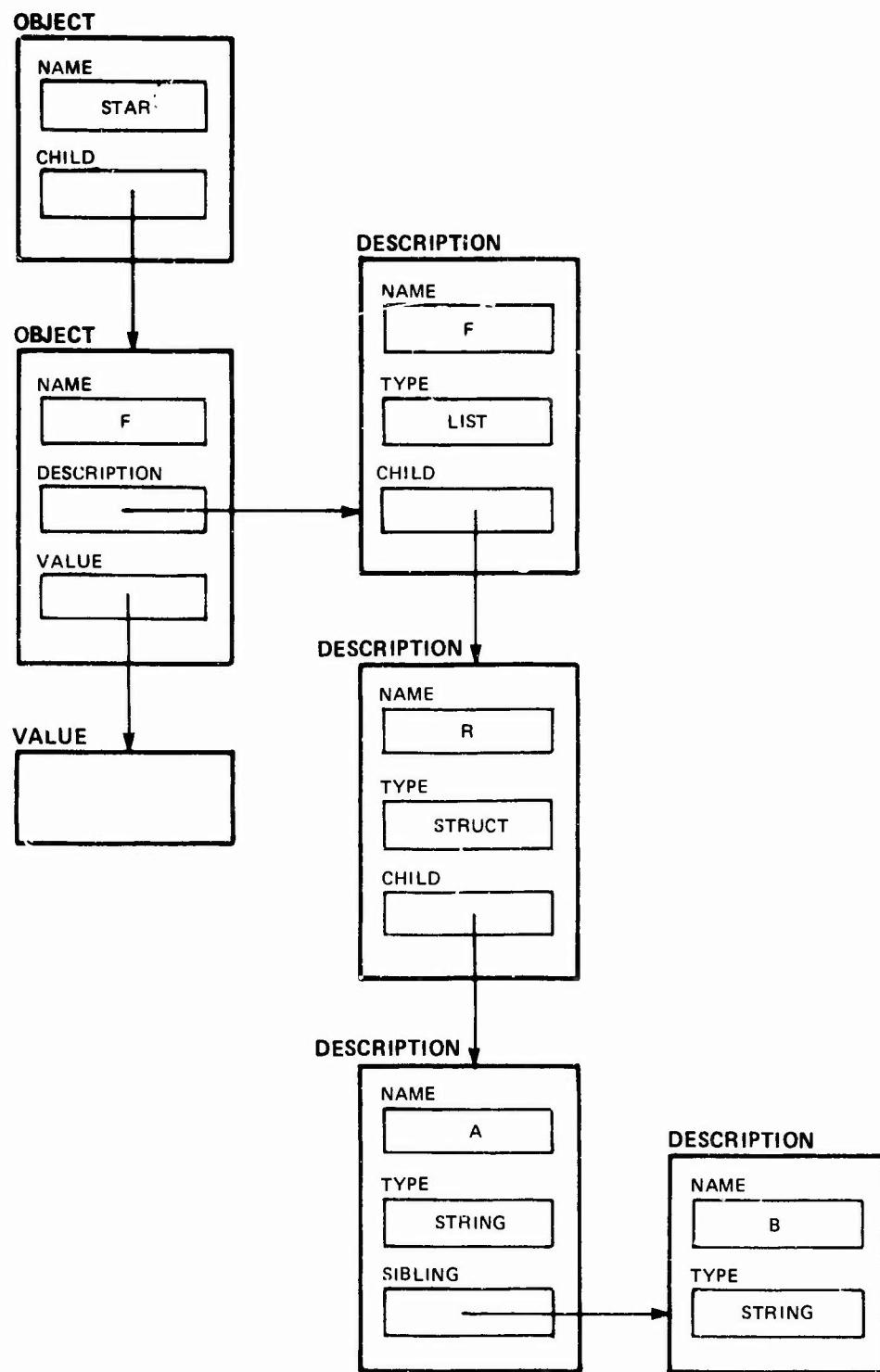
Figure 4-15
F immediately after creation

To add members to F, we need to use listops, and for this we must create two more objects: an object to represent the current member and an operation

descriptor (OPD). These are temporaries rather than permanent objects; they are also "top level" (i.e., not local to a request). Temporary, top level objects are created as members of the directory TOP/LEVEL. The calls to create them are:

```
L/CREATE(L/TOBJ(TOP/LEVEL),
                LD/STRUCT(M,
                        LD/STRING(A,FIXED,2),
                        LD/STRING(B,FIXED,2)))
L/CREATE(L/TOBJ(TOP/LEVEL),LD/OPD(OPF))
```

We create M to represent the current member; it's description is the same as the one input for a member of F (see the call which created F). The proper way to accomplish this is with a mechanism which shares the actual LIST member description with M; however, this mechanism does not yet exist in our model.

We now wish to add some data to F; each member will be a STRUCT containing two two-character STRINGs.

To begin the listop sequence:

```
L/LISTOP/BEGIN(L/POBJ(F),L/TOBJ(M),
                        L/TOBJ(OPF),ADD)
```

This call establishes the structure shown in figure 4-16. It initializes the OPD, making it point to F and M and recording that only the ADD suboperation is enabled.
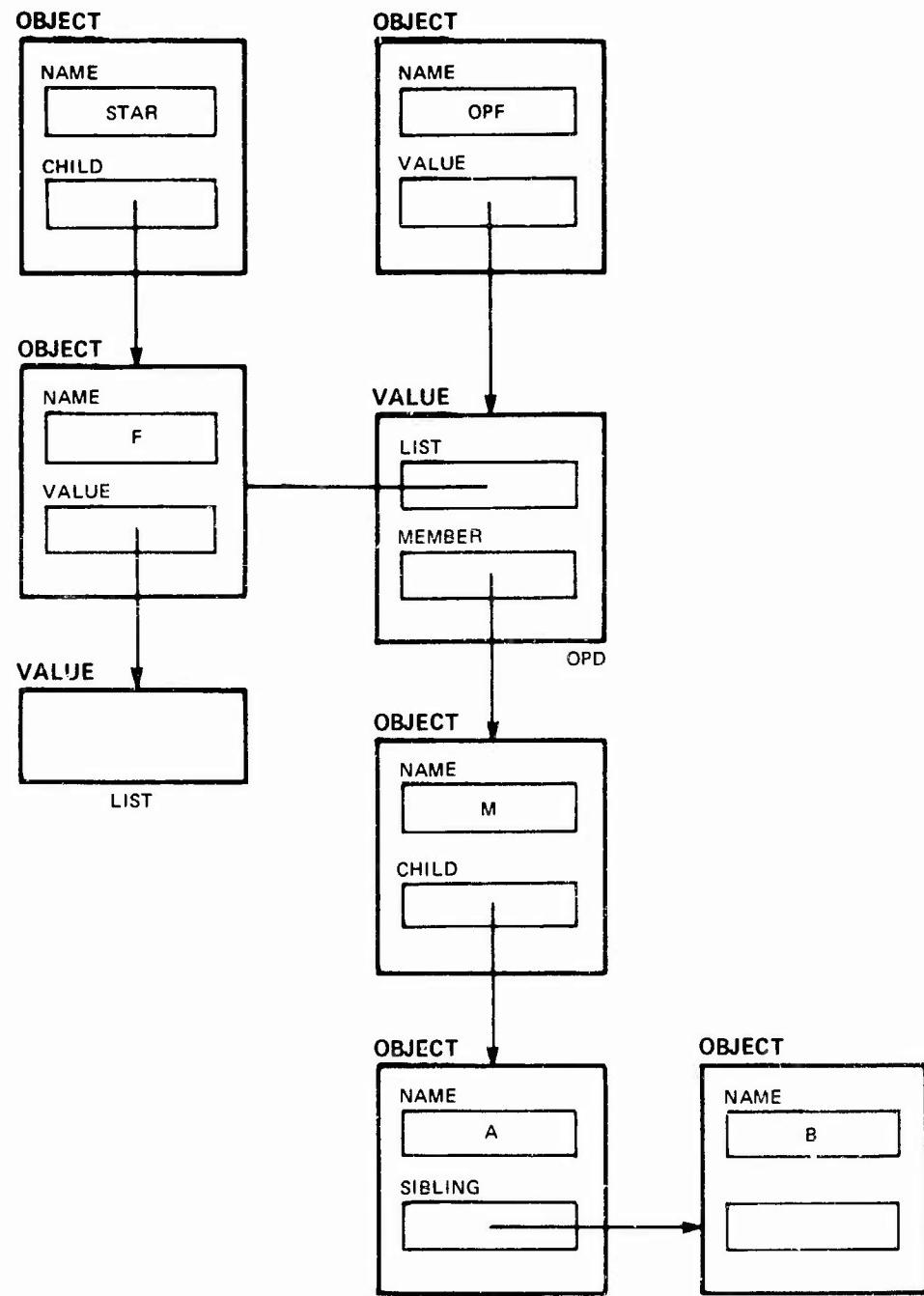
Figure 4-16
F, OPF and M after L/BEGIN/LISTOP

Next we must establish a current member. We want to add members to the end
(in this case, adding them anywhere would get the same effect, since the LIST
is empty), which is done by making LAST the current member.
L/WHICH/MEMBER(L/TOBJ(OP1),LAST)

Now, to add a new member to F, we can execute the following:
L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('AB'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('CD'))
L/CLOSE/MEMBER(L/TOBJ(OPF))

L/OPEN/MEMBER creates a STRUCT value for M. It does not affect the value of
F.   Each member of the STRUCT value is initialized when the STRUCT is
created. The result is shown in 4-17; notice that the STRUCT member values
are as yet unrelated to the objects M.A and M.B.

Figure 4-18 shows the changes accomplished by the first L/ASSIGN; the pointer
from the object M.A to the value was set up by a GET/STRUCT/MEMBER compiled
by L/TOBJ(M.A). The value was filled in by the assign operator. The second
assign has similar effect, filling in the second value.   The call to
L/CLOSE/MEMBER takes the value shown for M in 4-18 (with the second member
value filled in) and adds it to the value of F. The result is shown in 4-19;
compare with 4-16.
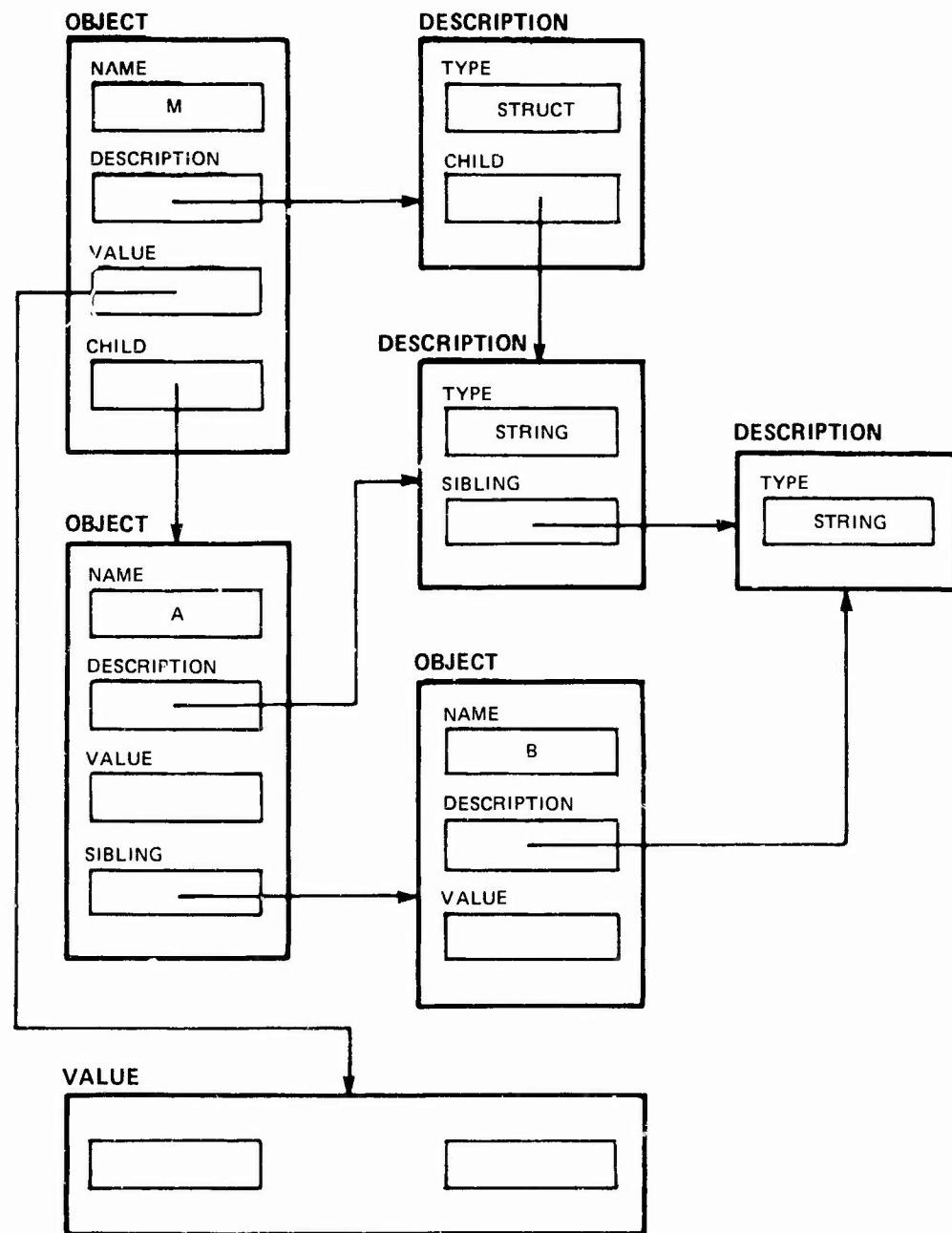
Figure 4-17
After L/OPEN/MEMBER
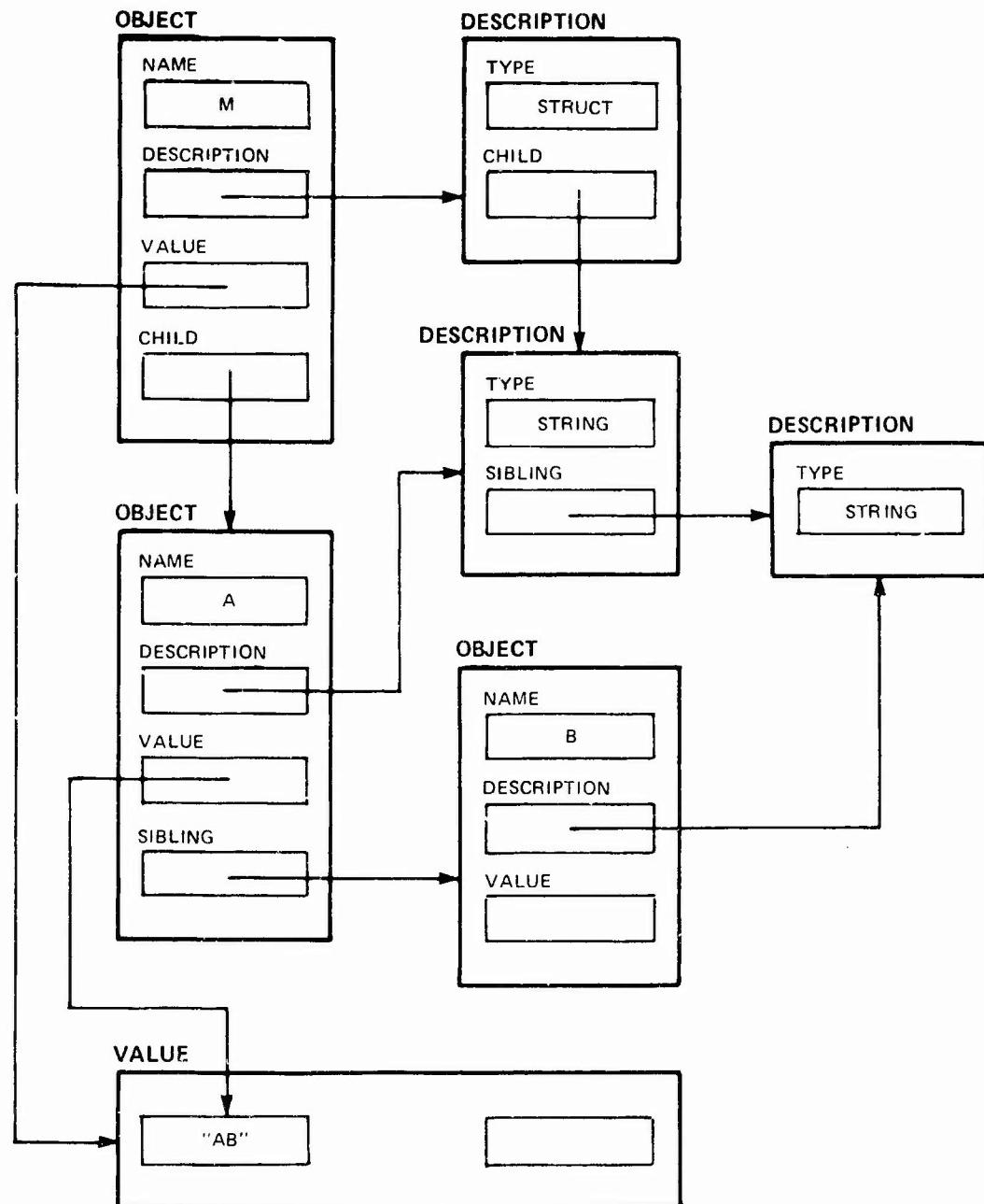
Figure 4-18
After first L/ASSIGN
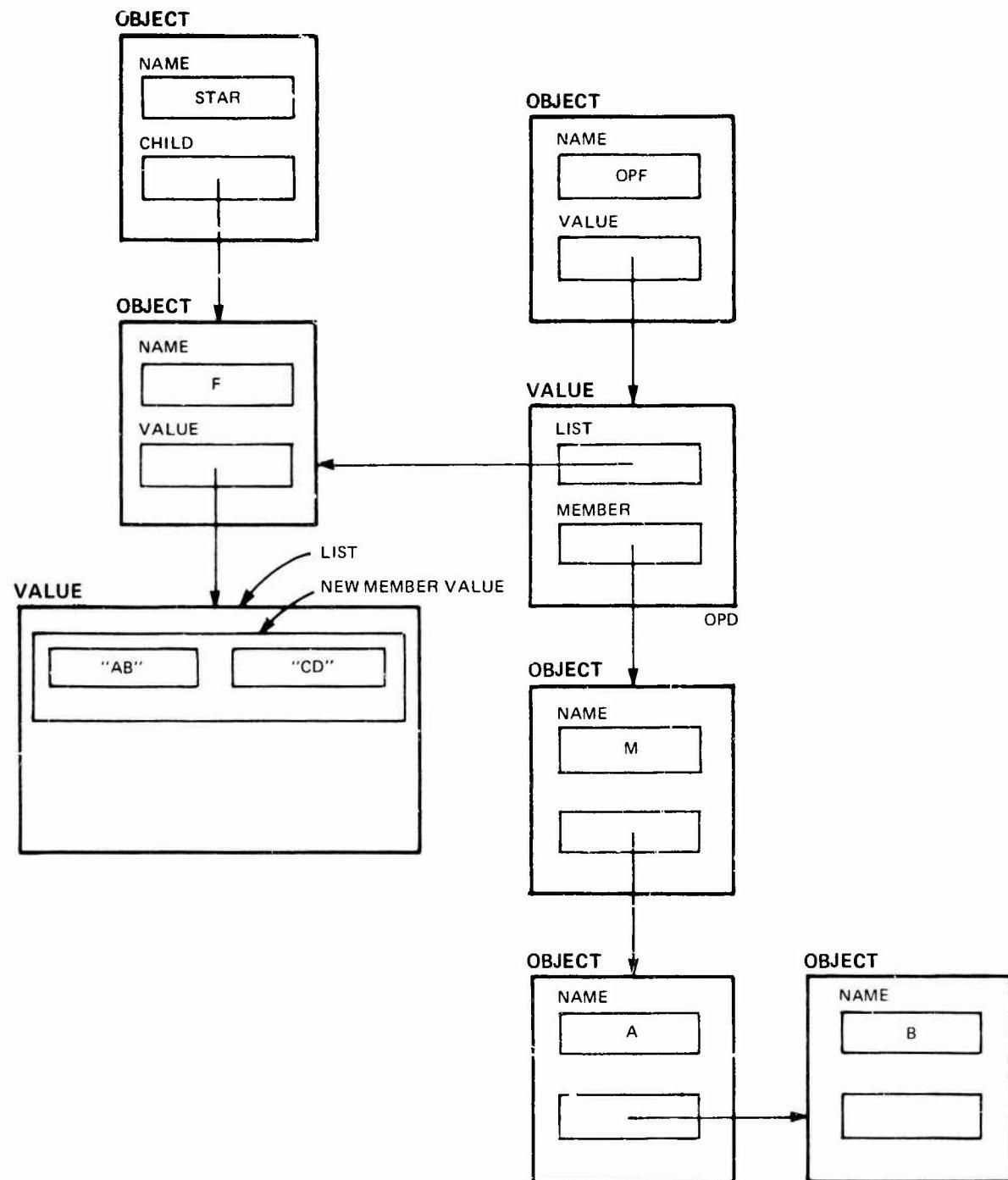
Figure 4-19
After L/CLOSE/MEMBER

By executing similar groups of four primitives, varying only values of constants, we can build up the LIST F shown in 4-20. The calls required are shown below:

```
L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('FF'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('GH'))
L/CLOSE/MEMBER(L/TOBJ(OPF))

L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('AB'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('IJ'))
L/CLOSE/MEMBER(L/TOBJ(OPF))

L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('CD'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('LM'))
L/CLOSE/MEMBER(L/TOBJ(OPF))
```

The add suboperation has the effect of making the member just added, the current member; thus no L/WHICH/MEMBER calls are needed in this sequence.

To terminate the sequence of listops:
```
L/END/LISTOP(L/TOBJ(OPF))
```

OBJECT

NAME

F

DESCRIPTION

VALUE

VALUE

"AB"   "CD"

"EF"   "GH"

"AB"   "IJ"

"CD"   "LM"
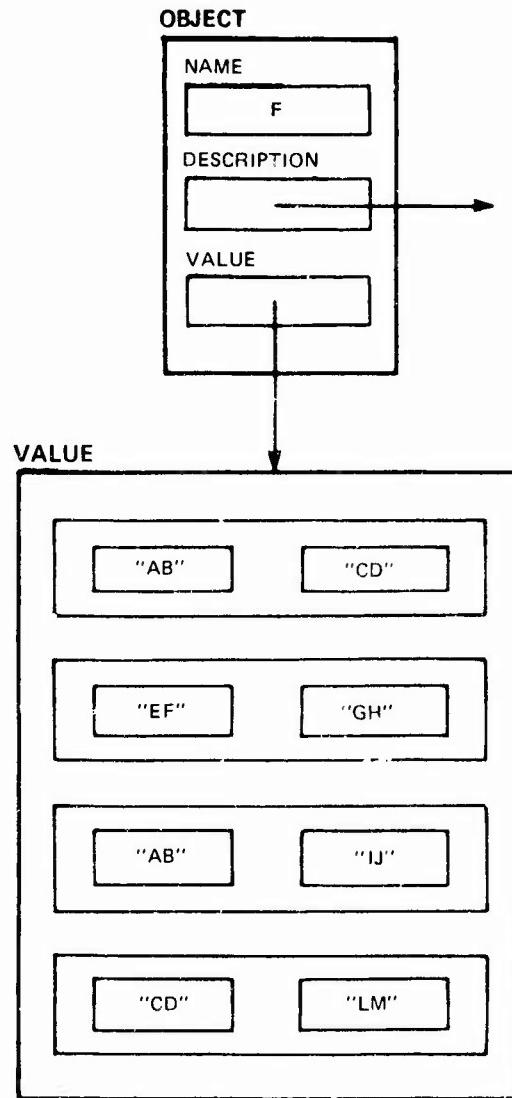
FIGURE 4-20
AFTER L/END/LISTOP

A slightly more interesting exercise is to construct calls which create a
LIST named G, having the same description as F, and then to copy into G all
members of F having A equal to 'AB'.

We must first create G, an OPD and an object to represent the current member.

```
L/CREATE("STAR",LD/LIST(G,
                         LD/STRUCT(R,
                                    LD/STRING(A,STRING,2),
                                    LD/STRING(B,STRING,2)))
L/CREATE(L/TOBJ(TOP/LEVEL),LD/OPD(OPG))
L/CREATE(L/TOBJ(TOP/LEVEL),LD/STRUCT(GM,
                                    LD/STRING(A,STRING,2),
                                    LD/STRING(B,STRING,2)))
```

We now need to initiate two sequences of listops, one on G and one on F.

```
L/BEGIN/LISTOP(L/POBJ(F),L/TOBJ(M),
                 L/TOBJ(OPF),GET)
L/BEGIN/LISTOP(L/POBJ(G),L/TOBJ(GM),
                 L/TOBJ(OPG),ADD)
L/WHICH/MEMBER(L/TOBJ(OPF),FIRST)
L/WHICH/MEMBER(L/TOBJ(OPG),LAST)
```

We will now sequence through the members of F; whenever the current member
has A equal to 'AB', we will add a member to G. We then copy the values of
the current member of F into the newly added member of G. When the current
member does not meet this criterion, we do nothing with it.

First, to write a loop that will execute until we get to the end of F:

```
L/TILL(L/END/OF/LIST(L/TOBJ(OPF)),x)
```

Whatever we put in this call to replace "x" will execute repeatedly until the
end/of/list flag has been set in OPF.

We must replace "x" with a single function call to in order to give L/TILL
what it is looking for. However, we will be executing "x" once for each
member of F, and will need to execute several listops each time.   The
solution is to use L/CF, the compound function function:

```
L/TILL(L/END/OF/LIST(L/TOBJ(OPF)),L/CF(y))
```

We can now replace "y" with a sequence of function calls.

Each time we iterate, we need to process a new member of F; initially we are
set up to get the first member. The following sequence, then, is needed:

```
L/CF(   L/OPEN/MEMBER(L/TOBJ(OPF),GET),
        z
        L/CLOSE/MEMBER(L/TOBJ(OPF)),
        L/WHICH/MEMBER(L/TOBJ(OPF),NEXT) )
```

The above is a compound function which will open the current member of F, do
something to it (represented above by "z"), close it, and ask for the next
member.

We want to replace "z" by a function call which tests the contents of A in
the current member of F, and either does nothing or adds a member to G,
copying the values of the current member of F. If "w" represents the action
of adding a member to G and copying the values, then we can express this:

```
L/IF(L/EQUAL(L/TOBJ(M.A),LC/STRING('AB')),w)
```

A suitable way to express "add a member and copy values" is:
L/CF(L/OPEN/MEMBER(L/TOBJ(OPG),ADD),
        L/ASSIGN(L/TOBJ(GM.A),L/TOBJ(M.A)),
        L/ASSIGN(L/TOBJ(GM.B),L/TOBJ(M.B)),
        L/CLOSE/MEMBER(L/TOBJ(OPG))

This is similar enough to the previous example so that no explanation should
be necessary.

Putting this all together, we get:
L/TILL(L/END/OF/LIST(L/TOBJ(OPF)),
   L/CF(L/OPEN/MEMBER(L/TOBJ(OPF),GET),
        L/IF(L/EQUAL(L/TOBJ(A),LC/STRING('AB')),
                L/CF(L/OPEN/MEMBER(L/TOBJ(OPG),ADD),
                        L/ASSIGN(L/TOBJ(GM.A),L/TOBJ(M.A)),
                        L/ASSIGN(L/TOBJ(GM.B),L/TOBJ(M.B)),
                        L/CLOSE/MEMBER(L/TOBJ(OPG))))
        L/CLOSE/MEMBER(L/TOBJ(OPF)),
        L/WHICH/MEMBER(L/TOBJ(OPF),NEXT)))

To conclude the operation, we execute:
L/LISTOP/END(L/TOBJ(OPG))
L/LISTOP/END(L/TOBJ(OPF))

The result is a LIST G whose first member has value ('AB','CD'), and whose
second member has value ('AB','IJ').   With a few variations on the above
example, quite a few LIST operations can be performed.

4.11 Higher level functions

While these primitive l/functions are useful, we would not ordinarily expect
users to operate in datalanguage at this low level. We want to make these
primitives available to users so that they can handle the exceptional case,
and so that they can construct their own high-level functions for atypical
applications. Ordinarily, they ought to operate at least at the level of the
following construction (which is legal in the real datalanguage currently
implemented):
 FOR G.R,F.R WITH A EQ 'AB'
        G.R=F.R
        END
This relatively concise expression accomplishes the same result as the
elaborate construction of l/functions given at the close of the preceding
section. We could define l/functions very similar to the semantic functions
used in the running software, and write the above request as:
 L/FOR(L/POBJ(G),R
        L/POBJ(F),R,L/WITH(L/EQUAL(L/TOBJ(A),
                                LC/STRING('AB')))
The differences between the l/function call and the datalanguage request
above it are principally syntactic.

In designing functions such as L/FOR and L/WITH, the central problems have to
do with choosing the right restrictions. One cannot have all the generality

available at the primitive level. Some important choices for these particular functions are: (1) handling multiple inputs and outputs, (2) when FORs are nested, how outer FORs restrict the options available to inner FORs, (3) generality of selection functions (may then in turn generate FORs?), (4) options with regard to where processing should start (are we updating, replacing or appending to the output list(s)?).

Finally, this problem is related to the more general problem of dealing with sets, which are a generalization of the idea of a collection of members in a LIST having common properties. FOR is only one of many operators that can input sets.

4.12 Conclusion

The present model, though embryonic, already contains enough primitives and data types to permit definition and generalized manipulation of hierarchical data structures. Common data management operations, such as retrieval by content and selective update can be expressed.

The use of this model in developing these primitives has resulted in precise, well-defined and internally consistent specifications for language elements and processing functions. Operating in the laboratory environment provided by the model seems to be a substantial benefit.

5. Further Work

In this section, we review what has been accomplished so far in the design and describe what work remains to be done before this design iteration of datalanguage is complete.

5.1 A Review

Most important among our accomplishments, we feel that we have delineated the problems and presented the broad outlines of a solution to development of a language for the datacomputer system. Key elements of our approach are the primacy of data desciption in capturing all the aspects of the data, the separation of logical and physical characteristics of data desciption, the ability of users to define different views of the same data, the ability to associate functions with different uses of data items, an attempt to capture common aspects of data at every possible level, and the ability of users to communicate with the datacomputer in as high a level as their application permits.

5.2 Topics for Further Research

Although more work needs to be done in general to turn out a finished design for datalanguage, we can single out certain issues which in particular need further investigation.

So far, only hierarchial data structures (i.e. those that can be modeled by physical containment) have been developed to any extent. We also intend to investigate and provide other types of data structures. We are confident that our language framework does not make assumptions that would prohibit such additions.

Our current work on access regulation centers on the use of multiple descriptions for data. We need to do more work on both the technical and administrative aspects of access regulation. Problems of encrypting data for both transmission and storage will also be investigated.

Another issue requiring further research is the protocol requirement for process interaction with the datacomputer.

Separation of the description into independent modules needs furhter research. In particular, we need to look into work which has already been done on separate specifications of logical descriptions, physical descriptions, and mappings between the two.

5.3 Datalanguage Syntax

We have not yet proposed a syntax for the datalanguage we are developing. Certainly the most difficult parts of the problem have been the semantic and pragmatic issues. We are confident that various syntactic forms can be chosen and implemented without excessive difficulty. It may be best to develop different syntactic forms for the language for different types of users or even for the various subparts of the language itself. As discussed in section 2, the user syntax for the datacomputer is supposed to be at a low level. It should be easy for <u>programs</u> to generate datalanguages requests in this syntax.

5.4 Further Work on the Datalanguage Model

The model provides an excellent foundation on which to build up a language with the facilities described in section 3. Much work is yet to be done.

For a while, emphasis will be on sets, high-level operators, language extension and data description.

We expect to model sets as a new datatype, whose value is ordinarily shared with other objects. Some further work on binding and sharing of values is needed to support this.

Sets can be regarded as a special case of generalized relations, which will come somewhat later.

High-level operators such as FOR will be constructed from the existing primitives, and will eventually be defined to have one effect but several possible expansions. The expansion will depend on the representation of the data and the presence of auxiliary structures.

Alternate expansions will be possible when the data description has been broken up into its various modules. This, also, requires some further research.

We feel that the language extension problem is much more easily attacked in the environment provided by the model datacomputer. In particular, we expect the laboratory environment to be helpful in evaluating the complex interactions and pervasive effects of operators in the language which extend the language.

Data description work in the near term will focus on the isolation of attributes, the representation of variable structure in description, the description of descriptions and the development of a sufficient set of builtin data types.

Later, we expect to model the semantics of pointers as a datatype, when the representation of the pointer and the semantics of the address space into which it points are specified in the description of the pointer.

A large number of lower-level issues will be attacked, including some of the problems discovered in the modeling to date. Some of these are pointed out in the discussions in section 4.

5.5 Applications Support

The datalanguage we are designing is intended to provide services to sub-systems
solving a broad class of problems related to data management. Examples of such
sub-systems are: report generators, online query systems for non-programmers,
document-handling systems, transaction processing systems, real-time data
collection systems, and library and bibliographic systems. There are many more.

The idea is that such systems will run on other machines, reference or store data
at the datacomputer, and make heavy use of datalanguage. Such a system would not
be written entirely in datalanguage, but a large component of its function would
be expressed in datalanguage requests; some controlling module would build the
requests and perform the non-datalanguage functions.

While we have experience with such applications in other environments, and we
talk to potential users, it will require some work to determine that our language
is actually adequate for them. That is, we are not attacking directly the
problem of building a human-oriented online query system; we are trying to
provide the tools which will make it easy to build one. There is a definite need
to analyze whether the tools are likely to be good enough. Of course, the
ultimate test will be in actual use, but we want to filter out as many problems
as we can before implementation.

An important component of supporting applications is that the using programs will
frequently be written in high-level languages such as FORTRAN, COBOL or PL/1. We
will want to investigate the ability of datalanguage to support such users, while
the design is taking shape.

5.6 Future Plans

This paper has laid the foundations for a new design of datalanguage. Section 3
provides an outline for a datalanguage design, which will be filled in during the
coming months. Following the issue of a detailed specification, we anticipate
extensive review, revisions, and incorporation into the implementation plans.
Implementation will occur in stages, compatible with the established plans for
development of datacomputer service and data management capabilities.